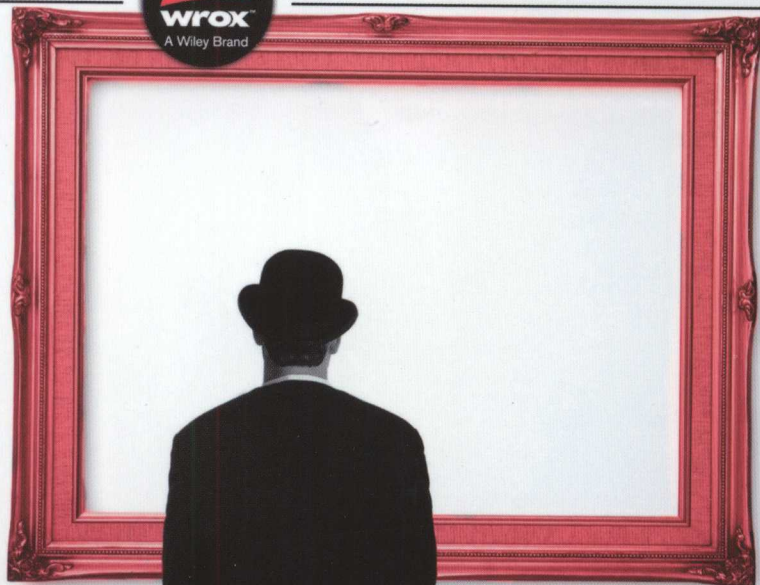


版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



Python Projects

Python

项目开发实战

[美] Laura Cassell 著
Alan Gauld 译
高弘扬 卫莹

清华大学出版社



Python 项目开发实战

[美] Laura Cassell 著
Alan Gauld 著
高弘扬 卫莹 译

清华大学出版社

北 京

Laura Cassell, Alan Gauld

Python Projects

EISBN: 978-1-118-90866-2

Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana

All Rights Reserved. This translation published under License.

Trademarks: Wiley, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Python is a registered trademark of Python Software Foundation Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2015-2354

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有 Wiley 公司防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

Python 项目开发实战 / (美) 卡塞尔 (Cassell, L.), (美) 高尔德 (Gauld, A.) 著; 高弘扬, 卫莹 译.
—北京: 清华大学出版社, 2015

书名原文: Python Projects

ISBN 978-7-302-41587-9

I. ①P… II. ①卡… ②高… ③高… ④卫… III. ①软件工具—程序设计 IV. ①TP311.56

中国版本图书馆 CIP 数据核字(2015)第 225360 号

责任编辑: 王 军 于 平

装帧设计: 牛静敏

责任校对: 成凤进

责任印制: 何 芊

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座

邮 编: 100084

社 总 机: 010-62770175

邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者: 北京富博印刷有限公司

装 订 者: 北京市密云县京文制本装订厂

经 销: 全国新华书店

开 本: 185mm×260mm

印 张: 21.25

字 数: 517 千字

版 次: 2015 年 10 月第 1 版

印 次: 2015 年 10 月第 1 次印刷

印 数: 1~3500

定 价: 59.80 元

产品编号: 062782-01

译者序

Python 是一门脚本语言。在面世之初，Python 并没有得到很多人的青睐。但是自从 2004 年以来，越来越多的公司发现了 Python 的强大之处。Python 的使用率也呈现线性增长。它甚至在 2011 年被评为年度编程语言。Python 语言简洁、易读、可扩展，可以用于很多方面。一些国外的科研机构采用 Python 作为开发语言。一些知名大学甚至开设了教授 Python 的教程，例如卡耐基梅隆大学和麻省理工学院。除了 Python 语言本身的特性之外，各种各样的扩展库也是 Python 的强大之处。这些丰富的扩展库可以帮助 Python 在各个领域都大显身手。Pypi 的出现更是为获得这些扩展库提供了极大的便利。

Python 的语言特性决定了它非常容易上手。市面上也已经有很多 Python 的入门书籍。这些入门书籍从语言本身出发，会详细介绍 Python 的一些重要特性。通过这些入门书籍，读者可以足够了解 Python 的基础。一些书籍会涉及 Python 的高级特性或一些额外的扩展包。然而，这些书并没有告诉读者：使用 Python 可以在实际中做什么。本书的最主要目的就是告诉 Python 初学者：Python 在真实世界中有哪些用途。然而在阅读本书之前，读者最好已经对 Python 有了初步的了解。本书循序渐进，逐步地为读者揭开 Python 在实际世界中的用途。

本书的第 1 章简要回顾了 Python 的核心知识。通过第 1 章，读者可以巩固 Python 的基础知识。接下来的一章主要介绍了 Python 的最重要的应用之一：脚本语言。Python 可以被用作脚本来协调操作系统上不同的应用并完成一个任务。第 3 章讲述了如何使用 Python 管理数据。第 4 章描述了如何使用 Python 创建桌面应用。在第 5 章中，Python 被用来开发网络应用。第 6 章讲述了如何在更大的项目中使用 Python。第 7 章探索了 Python 的前沿技术。

为了让读者能够更加深刻地理解书中所涉及的知识，作者在本书中插入了大量的示例。读者在使用本书时，应该动手试试这些示例。纸上学来终觉浅，要知此事须躬行。如果有条件，推荐读者去阅读感兴趣模块的源代码，这样可以快速地提高 Python 编程能力。

在此，非常感谢清华大学出版社的编辑们。没有她们的帮助，本书不可能顺利付梓。同时也感谢本书的其他译者。本书的全部章节由高弘扬、卫莹翻译，参与翻译的还有卫玲、刘扬、于洋、金宏兵、戴云、张国健、王辉、袁传清、张秀丽、高庆宝、陈守范。

在翻译中，译者本着信达雅的原则，反复揣摩唯恐不能完全表述作者的原文。然而介于译者能力有限，如有纰漏之处，在此向各位读者道歉，还望海涵，不吝指正。

译者

作者简介

Laura Cassell 从 1997 年就开始接触网络编程。在 21 世纪初，她自学了 Perl。那时，她发现编程资料急需改善和补充，但是教授编程的门槛却非常高。因此，她开始学习编程，这样就可以向更多人教授编程。

在乔治亚州的亚特兰大，**Laura** 创建了 **PyLadies Atlanta**，并开始为 **Big Nerd Ranch** 教授 **Python** 和 **JavaScript**。从此，她开始从事工程管理，并且现在定居波兰。在那里，她管理一组 **Python** 支持者，为 **New Relic** 有限公司做软件分析。同时，她在时间允许时会做义务教学和演讲。

Alan Gauld 是一位拥有超过 40 年 IT 从业经验的企业架构师。他主要的工作领域是电子通信和客户服务。他使用过的编程语言超过 20 种，创建的产品包罗万象，从大型机计费系统到嵌入式微控制器。在过去 15 年里，他主要的编程语言是 **Python**。他撰写过一本 **Python** 入门书籍，也是 **python-tutor** 邮件列表的联合负责人。

除了编程，他喜欢登山、徒步旅行和滑雪。同时，他也是一位摄影师、艺术家和声乐爱好者。他和妻子 **Heather** 生活在苏格兰。

本书目的

技术编辑简介

Alex Bradbury 是一位编译器黑客、Linux 极客和自由软件爱好者，他是 Raspberry Pi 项目的长期贡献者，同时也是 *Learning Python with Raspberry Pi* 的作者之一。现在，他是剑桥大学计算机实验室的一名研究员，也是 lowRISC 项目的创建者之一。lowRISC 是一个用来生产完全开源的单晶片系统(System-on-Chip, SoC)的非营利性项目。

Todd Shandelman 非常怀念在 IBM System/370 主机的打孔卡上用汇编语言编程的日子。多年来，他使用过多种软件技术(C、C++和 Perl 等)。现在，Todd 把他最多的精力都放在 Linux 命令行的 Python 2.x 和 Python 3.x 编程上。在空闲时间，他是一个俄语和希伯来语的专业译者，擅长外文排版设计，并且对处理 Unicode 和 UTF-8 很有心得。Todd 在纽约州立大学企业管理系获得理学学士学位。他和妻子、儿子一起生活在德克萨斯州休斯顿市。

致 谢

非常感谢 Alan Gauld，他对于本书做出了巨大贡献。感谢 Mary James 和 Jennifer Lynn，他们对本书提出了一些宝贵意见。因为有你们，本书质量才得以精益求精。

同样也感谢 Python 社区。迄今为止，你是我见到的最热情的社区。在社区中，人们可以感受到热情，可以接触到从专家到新手的每一个人。继续加油，感谢你让我成为这么优秀社区中的一员。

——Laura Cassell

非常感谢 Laura Cassell 推动了这个项目。感谢 Jennifer Lynn 一直激励着我们。感谢 Python 社区在过去 15 年的支持。

——Alan Gauld

前言

在某年的一次会议后，发到 PyLadies 组织者邮件列表的一封邮件问：“有人有兴趣写一本 Python 方面的书吗？”当时，我已经考虑撰写一本编程书很久了。在多年教课以及在 PyLadies 和其他编程聚会做指导之后，我意识到需要一本新的、特定类型的编程书。但是我并没有马上回复那封邮件。我知道写一本书是一项巨大的工程(确实是!)，会耗费我大量的时间和精力。我在周末和节假日也要工作(是的，我又对了!)。我也知道我一有一份教授编程课程的全职工作，同时也是乔治亚州亚特兰大本地 PyLadies 的主要组织者。我的孩子也会开始问我：“这个周末你要写书吗？”

以上内容都是真实的(实际上比我最初的想法还要多)，但是我知道书籍很重要。非常多的学生会在课后问我：“现在我已经了解 Python 基础知识了，我能做些什么呢？”我的答案总是：“你可以参与一些开源项目!”或者“参加一些 Python 高级课程”。但是这些答案既不能让他们满意也不能让我满意。正确答案应该是：“你必须真正地寻找一些事情来做：解决一个问题或实现一个需求”。这是因为，真正理解编程和一门编程语言的唯一方式就是用这门语言去解决问题。

然而，另一个问题又出现了：“我没有真正需要解决的问题”。所以，虽然我可以让我的学生去了解开源项目，而这实际上也非常有帮助，但是如果不了解技术，他们可能会迷失，甚至放弃。这样社区就又失去了一位可能带来有趣东西的程序员。所以，在与家人和朋友做大量交流之后，我意识到需要撰写这本书。

本书目的

多年以来，一直有人问我们，“我在熟悉 Python 基础后能做些什么？”，“我能学到什么？”，“我该何去何从？”。解决以上问题就是撰写本书的目的所在。

对于编程书籍来说，很多人都曾经历过的一个长期问题是它们都是从语言基础到深层概念。这些概念只有拥有计算机科学学位的人才能理解。但这并不酷！编程的大门应该向任何有兴趣的人敞开。我们都应该致力于降低编程的门槛。我们觉得 Python 做到了这一点，但是我们需要更进一步，并且开始理解人们是如何学习抽象想法和概念的，帮助他们学习编程。

可以将编程想象成学习如何盖房子，只知道需要木料，但是不知道如何用木料盖房子。你仍然需要理解结构工程、电气、水管设施、通风、高压交流电(High Voltage Alternating Current, HVAC)等。编程也是一样。语言只解释了盖房子需要木料。还有很多与木料相关的东西。我们希望帮助你了解这些概念。

本书读者对象

本书并不适合想要学习 Python 的初学者。实际上，作为本书的读者，你需要拥有一些 Python 编程基础。这意味着你已经学过一些教程。你也应该理解空格在 Python 中的作用、列表被包含在方括号([])中、但字典被包含在花括号({})中。本书适用于那些初学者，但应该已经学过一两个教程。这些人理解 Python 基础，但对 Python 可以实现的功能很感兴趣。

俗语说的好，需要是发明之母。在你学习编程时，这句话非常正确。如果你需要软件来执行特定函数或任务，那么围绕着需求学习一门语言就很容易。你有需求，语言就会帮助你，学习语言，解决问题，你学到了知识，并且立即付诸实践。这太棒了！然而，如果你觉得编程很有趣，但却没有需求，不知道要实现什么，结果会怎样呢？这就是本书要解决的问题。

本书会帮助你学习大部分人不会对初学者讲述的 Python 部分。书中涉及的大部分工具和技术只有在实践中才会遇到。然而，对于没有特定问题需要解决的新手程序员来说，学习这些工具可能比较困难。在很长一段时间里，没有人想要向开发者介绍这些工具，因为它们真的很常用。我们希望可以带你领略 Python 的能力和辉煌。

你将学习如何编写一个 Web 应用，以及如何使用 Python 库与数据库通信。如果你是一名系统管理员，还可以学到可以加速工作流的系统工具。我们将简要介绍诸如安全和最佳实践的话题，概述如何使用 Python 库创建图形用户界面(GUI)。还将介绍如何编写和使用应用编程接口(Application Programming Interfaces, API)，以及其他对 Python 程序员有用的话题。

本书内容简介

我们希望带你简要了解一下 Python 的基础知识，将向你介绍那些只有在解决问题时才会理解的概念。尽管我们不能在这里呈现所有将来可能需要解决的问题，但是我们可以为 Python 新手展示 Python 语言的强大特性和可以使用的包和技术。

首先，提供一个 Python 的速成课程，以防你已经忘记了所有东西。我们将复习基础知识，然后你可以决定是否完整阅读该章。接下来，将从脚本语言的角度来重新审视 Python。通过尝试使用 Python 编写一些小脚本来访问你的系统。这可以展示 Python 让你所拥有的非常基本的能力。之后会讨论数据，这其实就是编程的一切——操纵数据。你会使用 Python 提供的标准库来完成一些示例。我们甚至会讨论数据库，这样就可以对它有一个基本了解。我们想让你了解并接触系统中可能会接触到的每个部分。

在前三章之后，将介绍桌面应用。尽管这些在 Python 中并不常用，但也是语言的一个特性。在你的整个 Python 程序员的生涯中它都非常有用。接下来将介绍 Internet。这时，Python 会充当数据通信工具。你将学习有关 HTTP 和 Web 的所有知识，以及网站在

底层的工作方式，甚至会动手编写和使用 API。很多新手程序员对 API 都很迷惑。我们希望在本章揭开它神秘的面纱。

在最后几章，将介绍 Python 中一些更高级的话题，例如，如何在更大的项目中使用 Python、调试代码、创建测试模块、错误处理，以及创建自定义的异常和异常处理器。你在使用本书时、在将来查阅本书时、在使用 Python 编程时，都可以使用索引快速找到你想要的内容。

本书信息量很大，其中包含大量的工具和想法，可帮助你开始使用 Python。我们希望你可以自己动手尝试，并且花时间在你对感兴趣的概念和想法上做更多功课。在本书中，已经包含了大量实践练习来帮助你尝试新概念。在大多数章节中，还包含了一些挑战性内容，以帮助你巩固新知识。

使用本书须知

为了更好地使用本书，建议你所使用的现代计算机能够运行 Python 3.3 或更新版本，有一个能够舒适使用的好的文本编辑器，具有 Internet 连接(本书一些部分会使用)，以及足够的耐心和求知欲。我们也建议你使用 Internet 搜索任何遇到的问题。专业程序员实际上并不是什么都会。他们通常只知道那些每天需要处理的问题，他们的大部分时间都花在搜索和追踪问题发生的原因上。不要觉得依赖 Google 解决问题是很让人沮丧的。有时，使用 Google 搜索问题的能力和你的编程能力是同样重要的。

在使用本书的示例和项目时，你可能需要源代码。示例的源文件可以通过 Wrox 网站 www.wrox.com/go/pythonprojects 和 <http://www.tupwk.com.cn/downpage> 下载。

源代码

在完成本书示例时，可以选择手动输入所有代码，也可以使用本书附带的源代码。本书中用到的所有源代码都可以从 www.wrox.com 下载。对于本书，源代码下载的具体位置在 www.wrox.com/go/pythonprojects 的 Download Code 选项卡下。

可以在 www.wrox.com 搜索本书的 ISBN(本书的 ISBN 是 978-1-118-90866-2)来寻找代码。www.wrox.com/dynamic/books/download.aspx 上列出了当前所有 Wrox 书籍的完整代码下载列表。

www.wrox.com 上的大部分代码是使用 .ZIP、.RAR 或适用于当前平台的类似压缩格式压缩的。下载之后，使用合适的解压缩工具解压即可。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果你在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常

感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要在网站上找到本书的勘误表，可以登录 <http://www.wrox.com>，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

如果在 Book Errata 页面上没有看到你找出的错误，请进入 www.wrox.com/contact/techsupport.shtml，填写表单，发电子邮件，我们会检查你的信息，如果是正确的，就在本书的勘误表中粘贴一个消息，我们将在本书的后续版本中采用。

p2p.wrox.com

P2P 邮件列表是为作者和读者之间的讨论而建立的。读者可以在 p2p.wrox.com 上加入 P2P 论坛。该论坛是一个基于 Web 的系统，用于传送与 Wrox 图书相关的信息和相关技术，与其他读者和技术用户交流。该论坛提供了订阅功能，当论坛上有新帖子时，会给你发送你选择的主题。Wrox 作者、编辑和其他业界专家和读者都会在这个论坛上进行讨论。

在 <http://p2p.wrox.com> 上有许多不同的论坛，帮助读者阅读本书，在读者开发自己的应用程序时，也可以从这个论坛中获益。要加入这个论坛，必须执行下面的步骤：

- (1) 进入 <http://p2p.wrox.com>，单击 Register 链接。
- (2) 阅读其内容，单击 Agree 按钮。
- (3) 提供加入论坛所需的信息及愿意提供的可选信息，单击 Submit 按钮。
- (4) 然后就可以收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。



提示：不加入 P2P 也可以阅读论坛上的信息，但只有加入论坛后，才能发送自己的信息。

加入论坛后，就可以发送新信息，回应其他用户的帖子。可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息，可以在论坛列表中单击该论坛对应的 Subscribe to this Forum 图标。

对于如何使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作原理，以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。

目 录

第 1 章 Python 核心知识回顾	1
1.1 探索 Python 语言和解释器	1
1.2 回顾 Python 数据类型	3
1.2.1 数值类型：整数和浮点数	4
1.2.2 布尔类型	5
1.2.3 None 类型	6
1.2.4 容器类型	6
1.2.5 字符串	7
1.2.6 字节和字节数组	9
1.2.7 元组	10
1.2.8 列表	11
1.2.9 字典	12
1.2.10 集	13
1.3 使用 Python 控制结构	15
1.3.1 结构化你的程序	15
1.3.2 使用序列、块和注释	16
1.3.3 选择一个执行路径	17
1.3.4 迭代	18
1.3.5 异常处理	20
1.3.6 上下文管理	21
1.4 在 Python 中读取和输出数据	21
1.4.1 与用户交互	21
1.4.2 使用文本文件	23
1.5 扩展 Python	24
1.5.1 定义并使用函数	24
1.5.2 定义并使用类和对象	28
1.6 创建和使用模块和包	32
1.6.1 使用和创建模块	33
1.6.2 使用和创建包	34
1.7 创建示例包	35
1.8 使用第三方包	42

1.9 本章小结	43
第 2 章 Python 脚本	47
2.1 访问操作系统	48
2.1.1 获得关于用户和他们的 电脑的信息	49
2.1.2 获得当前进程信息	52
2.1.3 管理其他程序	54
2.1.4 更加高效地管理子进程	57
2.1.5 获取文件(和设备)的信息	60
2.1.6 浏览和操纵文件系统	62
2.1.7 探索目录树深度	68
2.2 使用日期和时间	71
2.2.1 使用 time 模块	71
2.2.2 datetime 模块介绍	74
2.2.3 calendar 模块介绍	75
2.3 处理常见的文件格式	76
2.3.1 使用逗号分隔的数值	76
2.3.2 使用 Config 文件	82
2.3.3 操作 XML 和 HTML 文件	85
2.4 使用 ctypes 和 pywin32 访问 原生 API	93
2.4.1 访问操作系统库	94
2.4.2 使用 COM 访问 Windows 应用	96
2.5 涉及多应用的自动化任务	97
2.5.1 使用 Python	98
2.5.2 使用操作系统工具	98
2.5.3 使用数据文件	98
2.5.4 使用第三方模块	98
2.5.5 通过命令行接口与子进程 交互	99

2.5.6	为基于服务器的应用使用 Web 服务	99	4.2.1	创建数据层	163
2.5.7	使用一个原生代码 API	99	4.2.2	创建核心逻辑层	165
2.5.8	使用 GUI 机器人学	99	4.2.3	创建用户界面	169
2.6	本章小结	100	4.3	使用 cmd 模块创建命令行界面	173
第 3 章	管理数据	103	4.4	读取命令行参数	175
3.1	使用 Python 存储数据	104	4.5	用一些对话框让命令行界面变得生动	176
3.1.1	使用 DBM 作为持久化字典	104	4.6	使用 Tkinter 编程 GUI	180
3.1.2	使用 Pickle 存取对象	109	4.7	创建简单的 GUI	183
3.1.3	使用 shelve 访问对象	111	4.8	创建 Tic-Tac-Toe GUI	186
3.2	使用 Python 分析数据	116	4.8.1	勾勒一个 UI 设计	186
3.2.1	使用 Python 的内置特性分析数据	116	4.8.2	创建菜单	187
3.2.2	使用 itertools 分析数据	119	4.8.3	创建 Tic-Tac-Toe 面板	188
3.2.3	使用 itertools 分析 LendyDB 数据	124	4.8.4	将 GUI 连接到游戏	189
3.3	使用 SQL 管理数据	126	4.9	扩展 Tkinter	194
3.3.1	关系型数据库的概念	126	4.9.1	使用 tix	194
3.3.2	结构化查询语言	127	4.9.2	使用 ttk	198
3.3.3	跨表链接数据	134	4.10	再次回顾借出库	199
3.3.4	多对多关系	140	4.11	探索其他 Python GUI 工具包	207
3.4	从 LendyDB 迁移到 SQL 数据库	143	4.11.1	wxPython	207
3.4.1	从 Python 访问 SQL	143	4.11.2	PyQt	208
3.4.2	创建 LendyDB SQL 数据库	145	4.11.3	PyGTK	209
3.4.3	插入测试数据	146	4.11.4	原生 GUI: Cocoa 和 PyWin32	209
3.4.4	创建一个 LendyDB API	148	4.11.5	Dabo	210
3.5	探索其他数据管理选择	154	4.12	存储本地数据	210
3.5.1	主从数据库	154	4.12.1	存储特定于应用的数据	211
3.5.2	NoSQL	155	4.12.2	存储用户选择偏好	211
3.5.3	云计算	155	4.12.3	存储应用状态	212
3.5.4	使用 RPy 进行数据分析	156	4.12.4	记录错误信息	212
3.6	本章小结	157	4.13	理解本地化	214
第 4 章	创建桌面应用	161	4.13.1	使用区域设置	214
4.1	组织应用程序	162	4.13.2	在 Python 中使用 Unicode	216
4.2	创建命令行界面	163	4.13.3	使用 gettext	218
			4.14	本章小结	220

第 5 章 Python 在 Web 中的应用.....223	
5.1 Python 在 Web 中的应用.....224	
5.1.1 Web 应用的组成部分.....225	
5.1.2 客户端-服务器关系.....226	
5.1.3 中间件和 MVC.....226	
5.1.4 HTTP 方法和头信息.....227	
5.1.5 什么是 API.....230	
5.2 使用 Python 进行 Web 编程.....234	
5.3 有关 Python 和 Web 的 更多知识.....247	
5.3.1 静态网站生成器.....247	
5.3.2 Web 框架.....247	
5.4 使用 Python 跨网工作.....248	
5.4.1 XML-RPC.....248	
5.4.2 套接字服务器.....249	
5.5 更多 Python 网络编程的 乐趣.....252	
5.6 本章小结.....253	
第 6 章 Python 在更大项目中的 应用.....255	
6.1 使用 doctest 模块测试.....256	
6.2 使用 unittest 模块测试.....261	
6.3 Python 中的测试驱动开发.....265	
6.4 调试 Python 代码.....266	
6.5 工作在更大的 Python 项目中.....275	
6.6 发布 Python 包.....279	
6.7 本章小结.....281	
第 7 章 探索 Python 前沿技术.....283	
7.1 使用 Python 绘图.....283	
7.1.1 使用 turtle graphics.....284	
7.1.2 使用 GUI Canvas 对象.....284	
7.1.3 绘制数据.....284	
7.1.4 使用 imghdr.....285	
7.1.5 Pillow 简介.....285	
7.1.6 试试 ImageMagick.....285	
7.2 使用 Python 辅助科学.....286	
7.2.1 SciPy 简介.....286	
7.2.2 使用 Python 辅助生物科学.....287	
7.2.3 使用 GIS.....287	
7.2.4 处理语言.....287	
7.2.5 综述.....288	
7.3 使用 Python 开发游戏.....288	
7.3.1 增强 PyGame 经验.....288	
7.3.2 探索其他选项.....289	
7.4 进入电影领域.....289	
7.5 与其他语言集成.....290	
7.5.1 Jython.....291	
7.5.2 IronPython.....291	
7.5.3 Cython.....292	
7.5.4 Tcl/Tk.....292	
7.6 进入物理领域.....293	
7.6.1 serial 选项介绍.....293	
7.6.2 RaspberryPi 编程.....294	
7.6.3 与 Arduino 对话.....294	
7.6.4 探索其他选项.....294	
7.7 创建 Python.....295	
7.7.1 修复 bug.....295	
7.7.2 文档化.....295	
7.7.3 测试.....295	
7.7.4 添加特性.....296	
7.7.5 参加会议.....296	
7.8 本章小结.....296	
附录 A 练习答案.....299	
附录 B Python 标准模块.....315	
附录 C 可用 Python 资源.....323	

1.1 探索 Python 语言和解释器

Python 是一个功能强大且拥有严格类型模型的高级语言。Python 是“胶水语言”的典范。

第 1 章

Python 核心知识回顾

本章主要内容:

- Python 语言的基本特性
- 如何使用 Python 模块机制
- 如何创建新模块
- 如何创建新包

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/pythonprojects 的 Download Code 选项卡下找到。第 1 章代码位于 Chapter 1 download, 每个文件都是根据本章提到的代码文件名命名的。

如果已经遗忘了一些基础知识, 本章开始部分提供了 Python 的简短回顾以及一些基础知识。本书的内容都是基于这部分基础知识的。如果对自己的 Python 编程基础有足够的信心, 则可以跳过这些内容直接去阅读感兴趣的部分。毕竟, 之后当需要温习这些内容时, 可以随时回到本章。

在本章, 你首先会了解 Python 的生态系统、数据类型和主要的控制结构, 以及函数和类的定义的知识。然后, 你会看到 Python 模块和包系统。最后会创建一个基础的新模块包。这个包会包含数个模块。

在本章的结尾, 你应该为进一步的内容做好准备, 并且开始将标准 Python 模块应用在真实的项目任务中。

1.1 探索 Python 语言和解释器

Python 是一个动态的但是拥有严格类型的编程语言(Python 是严格类型的编程语言,

解释器会追踪每个变量的类型)。Python 代码既被解释又被编译。Python 源代码首先被编译成字节码，然后被解释器解释。但是这个过程对于用户来说是透明的。你不需要显式地调用 Python 来编译你的代码。

Python 语言有几种实现版本。但最常用的版本是用 C 语言实现的，通常被称为 CPython。其他实现版本包括 Java 语言实现的 Jython，以及专门为微软.NET 平台实现的 IronPython。本书所用的 Python 实现版本是 CPython。



注意：在本书成文时，Python 存在两种版本流：2.x 版本和 3.x 版本。本书将专注于 3 版本。本书所涉及的代码已经在 3.x 版本流的几个版本中测试过，包括最新的 3.4 版本。当涉及与 2.x 版本的较大兼容性问题时，我们通常指的是 2.7 版本。

Python 程序通常被写在带有.py 后缀的文本文件中。Python 解释器被称为 python(小写)。实际上，Python 解释器并不在意文件的后缀，添加这个后缀仅仅是为了方便用户(在有些操作系统中，这样做也允许文件和解释器建立关联)。

也可以直接在解释器中输入 Python 代码。这个方法适用于高度交互的开发风格。在这种开发风格中，想法首先在解释器中形成原型或被测试，然后转移到代码编辑器中。当开始使用一个新概念或代码模块时，Python 解释器是一个强大的学习工具。

在这种模式下工作时，可以在操作系统命令提示符中输入 python 来启动解释器。系统会反馈给你一个包含 Python 版本和一些创建细节的信息。在这个信息的下面是一个交互式提示符。可在这里输入代码。它看起来如下所示：

```
ActivePython 3.3.2.0 (ActiveState Software Inc.) based on
Python 3.3.2 (default, Sep 16 2013, 23:10:06) [MSC v.1600 32 bit (Intel)]
on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

从这个信息中，我们知道解释器适用于 3.3.2.0 版本的 Python。它是一个 ActiveState 的发行版本(而不是 python.org 发行版本)。它是基于 Windows 32 位操作系统创建的。你获得的信息可能会稍有不同，但应该包含相同类型的信息。

如果想执行一个存储在文件中的程序而不是与 Python 解释器交互，则可以在操作系统提示符中使用 python 命令，并在后面附上文件的名称：

```
$ python myscript.py
```



注意：通常，也可以在你的文件管理器工具中双击文件，操作系统会调用 python 来自动运行程序。然而，这通常会导致：程序打开一个窗口，运行结束，然后在你看到结果前关闭窗口。所以你可能会倾向于在一个命令行提示中完整地输入 `python filename` 命令。

Python 带有两个非常有帮助的函数：`dir(name)`和 `help(name)`。它们有助于你学习和探索这门语言。`dir(name)`会告诉你 `name` 所确定的对象有哪些可用的名称。而 `help(name)`将会展示被称为 `name` 的对象的信息。当第一次导入一个新模块时，通常是不知道这个模块包含哪些函数或类。通过查看 `dir()`得到的模块列表，可以知道有哪些方法和成员是可用的。然后可以在列出的任意特性上调用 `help()`。一定要试试这些函数，它们是非常重要的信息来源。

1.2 回顾 Python 数据类型

Python 支持很多强大的数据类型。表面上看，这些数据类型和其他编程语言中对应的数据类型一样。但是在 Python 中，这些数据类型通常拥有强大的能力。任何东西在 Python 中都是对象，也因此拥有方法。这意味着可以对任何变量执行一系列的操作。内置的 `dir()`和 `help()`函数可以帮助你了解全部信息。在本节，你会看到标准数据类型以及它们最重要的操作。



提示：如果需要，Python 参考手册提供了全部细节(<http://docs.python.org/3.3/reference/>)。

需要注意 Python 的一些基础概念。首先，Python 变量仅是名称。变量名的创建是通过把类型的实例赋值给它们。变量本身并没有类型，而与它们绑定在一起的对象拥有类型。名称只是个标签，同样，它也可以被一个完全不同的对象重新赋值。赋值操作使用 `=` 操作符，所以把一个值赋给一个变量就如下所示：

```
aVariable = aValue
```

这段代码把值 `aValue` 绑定到变量名 `aVariable` 上。如果此变量名不存在，解释器会把这个名称添加到合适的命名空间中。

因此，在 Python 中区别变量和它指向的对象是非常重要的。可以用双等号(`==`)来检查两个变量是否相等，也可以用 `is` 操作符来检查两个变量的对象身份(也就是两个名称是否

指向同一个对象), 如下所示:

```
>>> aString = 'I love spam'
>>> anotherString = 'I love spam'

>>> anInt = 6
>>> intAlias = anInt
>>> aString == anotherString # test equality of value
True
>>> aString is anotherString # test object identity
False
>>> anInt == intAlias # same value
True
>>> anInt is intAlias # also same object identity
True
```

Python 根据你使用类型的方式对它们进行分类。比如, 所有类型都可以分类为可变的(mutable)或不可变的(immutable)。如果一个类型是不可变的, 它意味着这种类型的对象一旦创建后就不能再改变。可以创建一个新的数据项并把它赋值给同一个变量, 但是不能更改原来不可变的值。

Python 也支持多个容器类型, 有时也被称作序列(严格来讲, 容器是序列的子集, 稍后会清晰阐述它们之间的差别)。尽管并不是所有的序列都支持所有的操作, 但它们有一组共同的操作。

一些 Python 数据类型是可被调用的。这意味着可以像调用函数一样使用类型名来生成这种类型的一个新实例。如果没有给定初值, 则会返回一个默认值。你将在下面每个数据类型的描述中看到相关的示例。

现在, 你已经了解了操作 Python 数据类型的基础知识, 下面会看到不同的数据类型, 包括数值、布尔、None 类型以及各种容器类型。

1.2.1 数值类型: 整数和浮点数

Python 支持多种数值类型, 包括最基本的整数类型和浮点类型。

Python 整数类型的特点在于它在理论上是无限大的。事实上, 整数的大小只被你的计算机的内存限制。整数类型支持所有常用的数值操作, 比如加法、减法、乘法等。可以使用传统的中缀表示法进行算术运算。比如, 当相加两个整数时:

```
>>> 5 + 4
9
```

或:

```
>>> result = 12 + 8
>>> print (result)
20
```

整数的字面值会默认以十进制表示。可以通过在数值前面加 0 和进制的首字母作为前缀来使用其他进制。因此，二进制被表示为 0bnnn，八进制被表示为 0onnn，十六进制被表示为 0xnnn。

一个整数的类型是 `int`。对于浮点数和数字的字符串表示形式，如 '123'，可以用 `int` 从中创建整数。如下所示：

```
>>> int(5.0)
5
>>> int('123')
123
```

`int` 也可以通过设定第二个可选的参数进行非十进制到十进制的转换(不仅是二进制、八进制和十六进制，最高到三十六进制都可以)。如果想把一个十六进制数的字符串表示转换成一个十进制整数，可以使用：

```
>>> intValue = int('AB34',16)
43828
```

Python 浮点数的类型是 `float`。与 `int` 一样，可以用 `float()` 来转换字符串表示，比如把 '12.34' 转换成浮点数。也可以用它把整数转换成一个浮点数。与整数不同的是，`float()` 不能处理不同进制的字符串。

`float` 类型也支持常用的算术操作以及几种舍入选项。Python 的浮点数是基于美国电气与电子工程师协会(IEEE)标准并且拥有与底层计算机架构相同的取值范围。但与此同时，它们也存在着同样水平的精确度问题，这意味着进行浮点数的比较是非常冒险的。Python 提供了模块来处理固定精度小数(`decimal`)和有理分数(`fractions`)。这可以帮助缓解精度带来的问题。Python 也原生地支持复数(或虚数)数字类型，称为 `complex`。这些类型都有特定的应用场景，所以本书不做详细介绍。

1.2.2 布尔类型

Python 支持一种布尔类型 `bool`。它拥有两种字面值 `True` 和 `False`。`bool` 类型的默认值是 `False`。也就是说，调用 `bool()` 会产生 `False`。

Python 对其他类型也支持 `Truth` 和 `False` 的概念。例如，如果整数值是 0，那么它会被认为是 `False`，而其他所有的非 0 整数值都被认为是 `True`。这点对浮点数一样适用。0.0 被认为是 `False`，而其他所有的浮点数被认为是 `True`。

可以用 `int()` 把布尔数值转换成整数。`False` 会被转换成 0，而 `True` 会被转换成 1。

布尔类型拥有大部分你所期待的布尔代数运算，包括 `and`、`or` 和 `not`。但奇怪的是，Python 并不支持 `xor` 运算。



注意：布尔类型被实现为整数类型的子类。也就是说，布尔类型也支持一些你可能预料不到的操作，比如幂运算。可以输入像 `True**False` 的代码然后得到结果 1。但是你应该只是假装这些特性不存在，并且把它们当作实现的细节。否则，你的代码会变得特别混乱。

除了布尔类型，Python 也支持整数的按位布尔操作。也就是说，Python 会把两个整数的每一个位对当成布尔数值对，并且对每一个位对执行相应的操作。这些操作包括按位与(&)、或(|)、非(^)和异或(~)。还包括把二进制位组左移(<<)或右移(>>)的位移操作符。在本章，你还会看到更多关于按位操作的介绍。

1.2.3 None 类型

None 类型代表一个空对象。在 Python 环境中，只有一个 None 对象。所有对 None 的引用都使用同一个实例。这意味着与 None 的对象值相等测试通常会被对象身份测试代替，如下所示：

```
aVariable is None
```

而不是：

```
aVariable == None
```

None 是 Python 函数的默认返回值。在函数中，它经常作为默认参数的位置标记或标志位。None 是不可调用的，所以不能作为一个转换函数把其他类型转换为 None。在被当作一个布尔数值时，None 的值是 False。

1.2.4 容器类型

如上所述，Python 有几种用来表示不同容器或序列的类型。这些类型有字符串、字节、元组、列表、字典和集。你会在之后看到这些类型之间的相同点和不同点。标准库模块 `collections` 提供了其他一些特殊的容器类型。在接下来的叙述中，你偶尔会接触到它们。



注意：在接下来的讨论中，你会看到接受容器类型作为参数的操作。通常，这就包括 Python 中的可迭代变量。可迭代变量是指符合 Python 迭代协议的对象。简单说，可迭代变量就是可以在循环结构中使用的对象。大多数情况下，你不必担心它们。如果很有兴趣，可以在 Python 文档中详细了解。一个不错的学习起点是：<https://wiki.python.org/moin/Iterator>。

一些特性是所有集合共有的。为了避免在每种容器类型的讨论中都赘述这些特性，在此会介绍它们。

可以通过内置函数 `len()` 得到 Python 中任何集合的长度。这个函数接受一个集合对象作为参数，返回集合中元素的个数。

可以通过索引下标访问一个集合中的单个元素。可以通过在方括号中提供一个下标值(或为字典提供有效的键值)实现这一点。集合的下标从 0 开始，也可以通过使用负的下标实现从后往前索引。这样，集合最后一个元素的下标就是 -1。

虽然使用索引下标只能访问集合中的一个特定的元素，但可以用切片(slicing)来访问集合中的多个元素。切片操作包括一个开始下标、一个结束下标和一个步长。这三个数值通过冒号分隔。切片操作对于字典或集是无效的。步长参数可以帮助你实现比如每隔一个获取元素的操作。这几个参数都是可选的，默认值就是集合的开始下标、集合的最后一个元素下标和为 1 的步长。切片操作返回下标从 `start` 到 `end-1` 的所有(选择的)元素。

下面是在 Python 交互式提示中对字符串进行切片操作的示例：

```
>>> '0123456789'[:]  
'0123456789'  
>>> '0123456789'[3:]  
'3456789'  
>>> '0123456789'[:3]  
'012'  
>>> '0123456789'[3:7]  
'3456'  
>>> '0123456789'[3:7:2]  
'35'  
>>> '0123456789'[::-3]  
'0369'
```

可以使用 `sorted()` 函数对大多数集合进行排序。返回的结果是包含原集合元素的已排序列表。`sorted()` 的可选参数使元素的排序和排序顺序变得更加灵活。

通常，在布尔表达式中，空集合被认为是 `False`，反之是 `True`。有两个函数——`any()` 和 `all()`，对集合的真假判断进行了完善。它们有助于对集合的布尔值进行更加精确的判断。`any()` 函数接受一个集合作为参数，如果集合中的任何成员是 `True`，就返回 `True`。`all()` 函数接受一个集合作为参数，当且仅当集合中所有成员都为 `True` 时才返回 `True`。

1.2.5 字符串

Python 字符串本质上是 Unicode 字符的集合(使用 Unicode 的影响会在第 4 章中讨论)。默认的编码方式是 UTF8。如果你的工作语言环境是英语，大部分事情会如你所料的工作。一旦你开始使用非英语字符，事情就开始变得有趣了。目前，你将会在英语语言环境下工作，并一直使用 UTF8。

在 Python 中，字符串的字面值需要被引号封闭。Python 是非常灵活的。就这一点而言，Python 可以使用单引号('Joe')、双引号("Joe")、三重单引号('''Joe''')和三重双引号('"""Joe"""')

来界定一个字符串。很明显，引号的开头和结尾应该是同一种类型，但是任何其他类型的引号都可以被包含在字符串中。这对于撇号或其他的类似情况是尤其有用的(‘He said, “Hi!”’ 或 “My brother’s hat”)。任意类型的三重引号可以跨越多行。下面是一些示例：

```
>>> 'using single quotes'
'using single quotes'
>>> "using double quotes"
'using double quotes'
>>> print('''triple single quotes spanning
... multiple lines ''')
triple single quotes spanning
multiple lines
```

当一个字符串的字面值出现在模块、类或函数的开始并且没有赋值给任何变量时，系统会把它当成文档。当对这个对象调用内置函数 `help()` 时，这个字符串会作为输出的一部分。

当有特殊字符，比如制表符(`\t`)或换行符(`\n`)，在字符串中时，需要加一个反斜杠字符前缀。反斜杠的字面值也必须加这个前缀，这样它们看起来就像双反斜杠。也可以通过在整个字符串前面加一个字母 `r`(代表 `raw`)来避免加反斜杠前缀。这样做就表示特殊字符将不会被处理。不可打印的字符可以用一个反斜杠加上这个字符的十六进制代码在字符串中表示。例如，转义字符表示为 `\x1A`(注意：开头的 `0` 并没有出现在十六进制整数的字面值中)。

字符串是不可变的。这意味着一旦一个字符串形成之后，就不可以直接对它进行修改或添加。然而，可以基于现有的字符串，创建一个新的字符串。这就是许多 Python 字符串操作的工作原理。Python 支持范围广泛的字符串操作。这些操作大部分都实现为字符串类的方法。表 1-1 列出了一些最常用的字符串操作。

表 1-1 字符串操作

操 作	描 述
<code>+</code>	连接字符串。但是该操作在某种程度上是一个低效的操作。通常，可改用 <code>join()</code> 来避免使用它
<code>*</code>	乘法。该操作复制多个字符串并把它们连接起来
<code>upper</code> , <code>lower</code> , <code>capitalize</code>	这些操作改变字符串中字符的大小写
<code>center</code> , <code>ljust</code> , <code>rjust</code>	这些操作会将字符串按照需要在给定的字符宽度内进行对齐。空白部分会根据需要填充指定字符(默认是空格)
<code>startswith</code> , <code>endswith</code>	这些操作会检查子字符串是否匹配一行的开头或结尾。可选参数控制真正被测试的子部分。但这样会让操作名显得有些令人困惑。如果输入为多个子字符串组成的一个元组，它们也可以一次测试多个子字符串
<code>find</code> , <code>index</code> , <code>rfind</code>	这些操作返回找到子字符串的最低下标。如果查找失败， <code>find</code> 会返回 <code>-1</code> ，而 <code>index</code> 会抛出 <code>ValueError</code> 异常。 <code>rfind</code> 从字符串的右边开始查找。所以它会返回满足条件的最高下标

(续表)

操 作	描 述
isalpha, isdigit, isalnum, 等	这些操作检查字符串的内容。几个常用的测试类型分别是字母、数字和字母数字式字符。最常用的操作就是列出来的这几个
join	该操作连接一个字符串列表，并使用给定字符串作为分隔符。通常，一个单空格或无空格会作为分隔符来构建一个字符串。相比字符串连接操作，该操作更快也更节省内存
split, splitlines, partition	这些操作根据一个给定的分隔符将一个字符串分割成一个个子字符串列表(默认分隔符为空白，多个连续空格也算一个空白)。需要注意的是，在这个过程中，原字符串中的分隔符将会被删除。splitlines()可以高效地根据换行符分割并返回一个行列表。partition() 根据给定的分隔符分割一个字符串。但该操作只运行到第一次分割，然后会返回分割出来的第一个子字符串、分隔符和剩余的字符串
strip, lstrip, rstrip	这些操作从字符串的两端移除空白(默认)或指定的字符。lstrip ()只处理字符串的左边。rstrip()只处理字符串的右边。这些操作中，没有操作从字符串的中间移除空白。它们只是移除外围的字符。如果需要全局地移除字符，请使用 replace 操作
replace	该操作执行字符串替换。通过指定一个空字符串作为替代者，该操作可以高效地删除字符
format	该操作用来替换在 Python 版本 2 中使用的老版本的 C 语言 printf 风格的字符串格式化操作。在 Python 版本 3 中，printf 风格依然可用，但为了支持 format()，它已经被弃用了。Python 文档对字符串格式化有详细的解释。基本的概念是字符串中嵌入的大括号对形成了作为 format()参数传入的数据的占位符。大括号内可以包含可选的风格信息，比如填充字符(可以在本书中找到一些相关示例)

其他字符串操作也是可用的，只不过表 1-1 中的操作是最常用的。

在布尔表达式中，空字符串被看作是 False。所有其他的字符串都被看作是 True。

1.2.6 字节和字节数组

Python 支持两种面向字节的数据类型。一个字节是一个八位值，相当于一个范围在 0~255 之间的整数。它代表着计算机存储的或通过网络传输的原始的位模式。它们在使用上和字符串非常相似，并且支持很多相同的方法。这两个类型的名称分别是 byte 和 bytearray。

字节字符串的字面值被表示成被引号引起来，然后在前面加一个字母 b。字节字符串是不可改变的。字节数组与它类似，但它们是可改变的。

在实际中，你很少会用到字节字符串或字节数组，除非你要处理来自文件或网络的二进制数据。有一个问题可能会令你非常惊讶。如果用下标来访问单一的元素，返回值会是一个整数。这意味着把一个单一字符的字节字符串和一个索引下标的字符串值进行比较，产生的结果是 `False`。这个结果与对字符串进行相同操作产生的结果是不同的。下面是一个示例：

```
>>> s = b'Alphabet soup'
>>> c = b'A'
>>> s[0] == c
False
>>> s[0] == c[0]
True
```

可以看到，关键是在比较的两边都使用了下标索引。

可以使用 `struct` 模块将二进制数据从字节表示形式转换成正常的 Python 类型。当然，在完成该操作之前，首先还是要知道字节模式代表的是什么类型。

在布尔表达式中，空字节字符串被看作是 `False`。所有其他的字节字符串都被看作是 `True`。

1.2.7 元组

元组是任意对象的集合。既然这些对象被收集到一起，这就暗示着在它们之间可能存在逻辑关联。但 Python 语言并没有对元组包含的对象有任何的限制。Python 中的元组通常被形容为其他语言中记录或结构体的等价物。

元组的字面值包含一系列用逗号分开的值(或变量)。通常，为避免语义歧义，元组整体都被包括在圆括号中。但元组本身并没有这样的要求。

元组是不可改变的，所以一旦元组被创建之后，就不能修改或扩展它。可以像字符串一样，基于现有元组创建一个新的元组。可以通过调用 `tuple()` 类型函数来创建一个新的空元组。由于元组是不可改变的，因此它可以作为字典的键。

Python 元组有一个非常有用的特性，被称为解包(unpacking)。这个特性能够帮助你将元组中的值提取出来并赋值给单独的变量。最常在这样的场景见到这个特性：你想要将函数返回的元组中的值存储在不同的变量中。下面是一个使用 `divmod()` 函数的示例。`divmod()` 函数将整数除法得到的商和余数组成的元组作为返回值。

```
>>> print(divmod(12,7))
(1, 5)
>>> q,r = divmod(12,7)
>>> print (q)
1
>>> print (r)
5
```

注意，`q` 和 `r` 是如何被看作新的单值变量。

在 `collections` 模块中有一个 `namedtuple` 类。它允许通过名称(而不是位置)来索引元素。这一点结合了字典的一些优势和元组的紧凑性(节省内存)和不可变性。

在布尔表达式中, 空元组被看作是 `False`。所有其他元组都被看作是 `True`。

1.2.8 列表

列表在 Python 中是一种高度灵活的强大的数据结构。它们可以被用来模仿许多经典数据结构的行为, 也可以使用自定义对象类的形式作为其他数据结构的基础。它们是动态的, 与元组相似的是它们可以保存任何类型的对象, 与元组不同的是它们是可以改变的。所以可以修改它们的内容。也可以使用元组风格的解包, 把列表的元素赋值给单独的变量。

列表的字面值表示为包含在方括号内的、用逗号分隔开的对象序列。可以通过指定一对空方括号或使用 `list()` 类型函数的默认值来创建一个空列表。列表有很多方法来添加和移除成员。它们也支持一些算术风格的操作, 比如像字符串那样的连接和复制列表。

可以直接用值列表来初始化一个列表, 或以编程方式使用列表推导来构建一个列表。列表推导看起来就像是在列表方括号里面的一行 `for` 循环。下面的示例构建了一个 1~10 的偶数平方的列表:

```
>>> [n*n for n in range(1,11) if not n*n % 2]
[4, 16, 36, 64, 100]
```

表 1-2 列出了一些最常用的列表操作。

在布尔表达式中, 空列表被看作是 `False`。所有其他列表都被看作是 `True`。

表 1-2 列表操作

操 作	描 述
+	该操作连接两个列表
*	该操作会复制多个第一个列表。注意, 所有的复制对象都指向同一个对象。然而, 修改一个对象通常会导致奇怪的副作用。通常, 列表切片或列表推导是更好的选择
append	该操作添加一个元素到现有列表的尾端。新元素自己也可以是个列表。该操作可以高效地就地执行, 返回值是 <code>None</code>
extend	该操作添加一个列表的内容到另一个列表的尾端。它可以高效地连接两个列表。原列表被就地修改。返回值是 <code>None</code>
pop	该操作从列表的尾端或者根据提供的参数在指定位置移除一个元素。返回值是被移除的元素
index	该操作返回元素在列表中的第一个下标。如果在列表中没有发现这个元素, 会抛出 <code>ValueError</code> 异常(同名的字符串操作也有类似的行为)
count	该操作返回指定元素在列表中的个数
insert	该操作在指定下标前插入一个元素。如果这个下标过大而超出列表范围, 这个元素会被添加到列表的尾端

(续表)

操 作	描 述
remove	该操作移除列表中的第一个指定元素。如果在列表中没有这个元素，会抛出 ValueError 异常
reverse	该操作原地反转列表中的元素
sort	该操作对列表中的元素进行原地排序。可选参数为如何进行排序提供了灵活性。如果想得到一个排好序的列表副本并且不修改原列表，则可以使用 sorted() 函数

1.2.9 字典

字典是初学者经常忽略的超级强大的数据结构。它为大量常见的编程问题提供了解决方案。字典使用起来就像列表，但是它的元素是通过键值(而不是数值索引)的方式被访问的。因此，一个 Python 字典就是一个(无序的)键值对序列。

键可以是任意不可改变的值，包括元组。键在字典中必须是唯一的。值可以是任意的 Python 对象，包括另一个字典、一个列表或者其他任何 Python 认为是对象的东西。

字典是高度优化的，所以字典的访问时间是非常快的。实际上，Python 内部的其他部分也广泛地使用字典，包括实现命名空间和类。字典也为这种情况在任何地方提供了一种解决方案：动态命名的值需要被存储起来并支持访问。字典的键不是连续的，这一点也让字典很高效。因为 Python 使用了哈希算法将键映射到一个稀疏数组结构中(如果不理解最后一句话，不必担心，很多人都不理解，但是你真的不需要理解它。对你而言，这句话意味着 Python 字典很快，而且在内存的使用上也很高效)。

字典的字面值包含逗号分隔的键值对。键和它对应的值用冒号分隔，然后整体被包含在一对花括号或 {} 中。它看起来如下所示：

```
>>> {'aKey': 'avalue', 2: 7, 'booleans': {False: 0, True: 1}}
{'aKey': 'avalue', 2: 7, 'booleans': {False: 0, True: 1}}
```

可通过键值而不是数值索引的方式来访问字典中存储的值。如果之前的示例被存放在一个名为 D 的变量中，可按如下方式访问键为 aKey 和 True 的值：

```
>>> D['aKey']
'avalue'
>>> D['booleans'][True]
1
```

可以使用一对空的花括号或 dict() 类型函数的默认值来创建一个空字典。

字典有一些额外的操作可用来抽取键列表和值列表，以及处理默认值。表 1-3 描述了一些操作。

由于字典的实现本质，字典是无序的。确实，当新数据被插入时，顺序可能发生改变。collections 模块有一个 OrderedDict 类。如果需要的话，它可以维持插入的顺序。如果键是

可比较的，`sorted()`函数会返回一个已排序的键列表。如果键不可比较(就像前面示例那样)，`sorted()`方法会抛出一个 `TypeError` 异常。

`collections` 模块也提供了一个 `defaultdict` 类。这个类能帮助你指定一个默认值。在任何情况下使用一个不存在的键时，它都会为这个给定的键用默认值创建一个新元素。这与之前描述的 `setdefault` 方法很相似。这可能是个喜忧参半的事情，因为它可能会因为糟糕拼写的键产生虚假的条目。

在布尔表达式中，空字典被看成是 `False`。所有其他字典都被看作是 `True`。

表 1-3 字典操作

操 作	描 述
keys, values, items	这些方法返回类似列表的对象(称为字典视图)，分别包含键、值和键值元组。这些视图是动态的，所以任何针对字典的改动(删除操作等)在它们创建之后还是会反映在视图中
get, pop	这些方法接受一个键和一个可选的默认值作为参数。如果键存在，那么 <code>get</code> 方法从字典中返回键对应的值。如果键不在字典中，则返回这个默认值。 <code>pop</code> 方法以同样的方式工作，但是如果键存在，则会把它从字典中删除。 <code>get</code> 有一个默认值 <code>None</code> 。但是如果没有给出默认值， <code>pop</code> 会抛出一个 <code>KeyError</code> 异常
setdefault	该操作的表现很像 <code>get</code> 。但是，如果键不在字典中，它会用给定的键和默认值在字典中创建一个新的键值对
fromkeys	该操作使用一个序列来提供键和一个给定的默认值(如果没有给定值，则默认值是 <code>None</code>)来初始化一个字典。该操作通常会直接调用 <code>dict.fromkeys()</code> ，而不是在一个现有的字典上调用它

1.2.10 集

集(Set)体现了一个在编程中常用的数学概念：一个集中的元素必须是唯一的。在 Python 中，集和有键但无对应值的字典有很多相似的地方。

集在 Python 的类型是 `set`。对于字典键的基本规则同样也适用于集。也就是说，集里的值必须是不可变的并且是唯一的(的确这也是集的精髓)。`set()`函数的默认返回值是空集。这也是唯一能表达一个空集的方式，因为 `{}` 已经被用来表示一个空字典。`set()`函数接受任意类型的集合(collection)作为参数，并且把它转换为一个集(字典的值会丢失)。

在 Python 中还有一种集类型，称为 `frozenset`。它是不可变的，并且基本上是一个只读集。它的构造函数和 `set()`一样工作，但是它只支持一部分集操作。因为 `frozenset` 是不可变的，所以可以使用 `frozenset` 作为一个普通集的元素。

集的字面值表示为用花括号将逗号分隔的元素括起来：

```
myset = {1,2,3,4,5}
```

集不支持索引或分片操作。和字典一样，集没有任何固有的顺序。`sorted()`函数返回一个有序的值序列。集有一系列数学风格的集操作。这些操作是其他集合所没有的。表 1-4 列出了 `set` 类型和 `frozenset` 类型都支持的操作。

表 1-4 集操作

操 作	描 述
<code>in</code>	该操作检查单一元素是否在集中。注意：如果被测试的元素本身是一个集， <code>S1</code> ，当且仅当 <code>S1</code> 作为一个集是目标集的元素时，结果才是真。它与 <code>subset()</code> 测试是不同的
<code>issubset</code> , <code><=</code> , <code><</code>	这些测试检查一个集是不是目标集的子集。被测试集的所有元素是否都是目标集的元素。如果两个集是完全相同的，前两个测试的结果会返回 <code>True</code> ，而 <code><</code> 操作符的测试会返回 <code>False</code>
<code>issuperset</code> , <code>>=</code> , <code>></code>	这些测试检查一个集是否是另一集的超集。目标集的所有元素是否都是源集的元素。如果两个集是完全相等的，那么前两个测试的结果会返回 <code>True</code> ，而最后一个操作会返回 <code>False</code>
<code>union</code> , <code> </code>	这些操作返回两个(或更多)集的并集。 <code>union</code> 方法接受一个逗号分隔的集列表作为参数，但是操作符的用法是将它插在集之间
<code>intersection</code> , <code>&</code>	这些操作返回两个(或更多)集的交集。这个用法与 <code>union()</code> 相似
<code>difference</code> , <code>-</code>	这些操作会返回那些在源集中但不在目标集中的元素
<code>symmetric_difference</code> , <code>^</code>	返回不在两个集的交集中的所有元素。这个方法只能应用于两个集。但 <code>^</code> 操作符可以通过中缀风格被应用在多个集上

需要注意的是，表 1-4 中的方法衍生形式可以接受任意容器类型作为参数，但是中缀操作只能作用在集上。

表 1-5 列出了只支持集的修饰符操作。尽管 `frozenset` 可以作为其中几个操作的参数，但这些操作不能被用在 `frozenset` 上。注意：这些操作会修改源集本身，它们并不返回一个集，而是返回 Python 的默认值 `None`。这里面的中缀操作只作用于两个集(不像表 1-4 中的操作)，并且只支持真正的集，不支持其他容器类型。可以在多个集上使用这些方法。在需要时，其他容器类型会被转换成集。

在布尔表达式中，空集被看作是 `False`。其他集都被看作是 `True`。

在下一节中，当探索 Python 提供的不同的控制结构时，你会在代码中用到数据类型。

表 1-5 集修饰符操作

操 作	描 述
<code>update</code> , <code> =</code>	这些操作会把目标集(或多个集)的元素添加到源集中
<code>intersection_update</code> , <code>&=</code>	除了在源集和目标集的交集的元素，这些操作会移除其他所有元素。如果涉及多于两个集，该操作的结果是所有涉及的集的交集

(续表)

操 作	描 述
difference_update, -=	这些操作会移除所有在交集集中的元素。如果涉及多个集，被移除的元素就是在源集与其他任意集的交集集中的元素
symmetric_difference_update , ^=	这些操作返回除了在交集集中的两个集的值。注意，该操作一次只能作用于两个集
add	该操作把给定的元素添加到集中
remove	该操作把指定元素从集中移除。如果没有发现这个元素，它会抛出一个 KeyError 异常
discard	如果给定的元素存在，该操作会把它从集中移除。如果元素不存在，该操作不会抛出 KeyError 异常
pop	该操作会随机地移除一个元素，并返回这个元素。如果集为空，它会抛出一个 KeyError 异常
clear	该操作会清空集中的所有元素

1.3 使用 Python 控制结构

在这一节，你会首先看到一个 Python 程序的整体结构，然后了解到每个基础的控制结构，包括顺序、选择和迭代。最后，你会看到 Python 如何处理错误，回顾上下文管理器，以及学习如何与外部世界交换数据。

1.3.1 结构化你的程序

Python 程序并没有任何必需的预定义入口(比如一个 main()函数)。它们的表现形式就是一个文本文件中的源代码。程序从文件的顶端开始被顺序地读入和执行(定义，比如函数，被执行的方式是函数被创建然后把它赋值给一个名称。但函数内部的代码在函数被调用之前并不执行)。

Python 并没有任何特别的语义来表明一个源文件是一个程序还是一个模块。而且可以看到，一个给定的文件既可以作为程序也可以作为模块使用。一个经典的可执行程序文件包括一系列重要的用于导入任何需要的代码模块的语句，一些函数和类的定义，以及一些可以直接执行的代码。

在实际中，对于一个重要的程序来说，大部分函数和类定义都会放在模块文件中，然后被包含在导入中。这在启动应用程序时添加了一个很短的驱动代码。通常这个代码会放在一个函数中，而这个函数常被命名为 main()。但这纯粹只是为了遵循编程惯例，并非是 Python 要求的。

最后，这个 main 函数需要被调用。通常在主脚本的末端会放置一个特殊的 if 语句来

调用这个 `main` 函数。如下所示：

```
if __name__ == "__main__":  
    main()
```

当 Python 检测到一个程序文件正在被解释器执行而不是作为一个被导入的模块，它会把特殊变量 `__name__` (注意两边都是双下划线) 设置为 `"__main__"`。这意味着任何在 `if` 代码块内的代码只有在这种情况下才会被执行：脚本被作为主程序运行而不是作为被另一个程序导入的文件。如果这个文件本来只是想被用作模块的话，`main()` 函数可能会被 `test()` 函数替换。`test()` 函数会执行一些列的单元测试。再次强调下，实际使用的函数名对 Python 来说是无关紧要的。

1.3.2 使用序列、块和注释

最基础的编程结构就是一个语句序列。通常，Python 语句会单独占据一行，所以一个序列也仅仅是一系列的行。

```
x = 2  
y = 7  
z = 9
```

在这个示例中，所有语句都是赋值操作。其他有效的语句包括函数调用、模块导入或定义。定义包括函数定义和类定义。下面讨论的各种各样的控制结构也是有效的语句。



注意：Python 允许你在一行中包含多个语句，但是这些语句之间要用分号分隔。因此，下面这行代码包含了三个语句：

```
x = 2; y = 7; z = 9
```

Python 社区不推荐使用这种风格，更倾向于一个语句单独一行。

Python 是一个块结构语言，并且程序块是通过缩进级别指示的。缩进的数量是非常灵活的。尽管大多数 Python 程序员坚持使用三个或四个空格来最优化可读性，但 Python 本身不介意这个。不同的集成开发环境 (IDE) 和文本编辑器对如何使用缩进有它们自己的理解。如果使用多种编程工具，你可能发现你会因为编程工具使用了不同的制表符和空格组合而得到缩进错误报告。如果可能的话，设置你的编辑器来使用空格而不是制表符。

缩进原则在注释上是个例外。一个 Python 注释以一个 `#` 符号开始一直到这行的结束。不管当前的缩进级别，Python 接受在一行中的任何位置开始注释。但是根据惯例，即使是注释，程序员也倾向于同样保持缩进的级别。

1.3.3 选择一个执行路径

Python 支持有限选项的选择。最基础的结构是 `if/elif/else` 构造。`elif` 和 `else` 部分是可选的。它看起来如下所示：

```
if pages < 9:
    print("It's too short")
elif pages > 99:
    print("It's too long")
else: print("Perfect")
```

注意在每个测试表达式末端的冒号。这是 Python 中的指示器，它用来指示一个新的代码块即将来临。它没有开始和结束标记(比如 `{}`)，冒号是唯一的标示。如果它包含的代码块只有一行，那么可以把它放在与冒号相同的行中，否则，它必须作为一个缩进的代码块。即使是只包含一行代码，许多 Python 程序员也更喜欢使用缩进的代码块风格。

同样需要注意的是，可以有任意多个 `elif` 测试表达式，但是至多只可以有一个 `else` 子句，没有也可以。

另一个在 Python 中的你能发现的选择结构是条件表达式选择器。它根据给定的测试条件从多个值中产生一个。它看起来如下所示：

```
<a value> if <an expression> else <another value>
```

一个示例是一个屏幕坐标一直被加 1 直到某一极限(可能是屏幕的最大分辨率)，然后被重置为 0。这可以被写为：

```
coord = coord + increment if coord < limit else 0
```

与之相对等的更传统的写法是：

```
if coord < limit:
    coord += increment
else:
    coord = 0
```

你应该非常小心地使用条件表达式选择器，因为它非常容易产生晦涩的代码。如果不确定的话，则应该使用扩展的 `if/else` 形式。

最终，还有一点值得注意的是 Python 的比较表达式。在许多编程语言中，如果想要测试一个值是否在两个限定值之间，则需要两个单独的测试，如下所示：

```
if aValue < upperLimit and aValue > lowerLimit:
    # do something here
```

Python 会非常乐意看到这样的代码，但是它也提供了一个有用的捷径。可以把两个比较表达式按如下方式联合起来：

```
if lowerLimit < aValue < upperLimit:
    # do something here
```


1.3.4 迭代

Python 提供了几种迭代的方法。最基本也最常见的就是 `while` 循环。`while` 循环如下所示：

```
while BooleanExpression:
    aBlockOfCode
else:
    anotherBlock
```

注意在 `while` 语句的末端有一个冒号(:)。这表示在它之后跟着代码块。也要注意代码块的缩进。原则上,只要 `BooleanExpression` 的值还是 `True`, 代码块就会被执行。然而,有两种方法可以忽略 `BooleanExpression` 的值,直接退出 `while` 循环。它们是 `break` 语句和 `return` 语句。`break` 语句会立刻退出循环。如果循环是在一个函数定义中, `return` 语句也可以起到这个作用。`return` 语句会立刻退出函数, 所以也会退出函数内的任意循环。

`else` 子句是可选的, 而且在实际中也很少被用到。它会在 `BooleanExpression` 为 `False` 时执行, 包括循环正常退出的情况。如果循环是被 `break` 或 `return` 语句退出的, 则 `else` 子句就不会被执行。

`while` 循环的一个惯用语法是把 `True` 作为测试条件来创建一个无限的循环, 然后在循环体内部放置一个 `break` 测试。在下面的示例中, 循环会读取用户命令并处理它们。如果命令包含字母 `q`, 循环就会退出。

```
while True:
    command = input('Enter a command[rwq]: ')
    if 'q' in command.lower(): break
    if command.lower() == 'r':
        # process 'r'
    elif command.lower() == 'w':
        # process 'w'
    else:
        print('Invalid command, try again')
```

`break` 有一个名为 `continue` 的同伴语句。`break` 会退出语句块和循环, 然而 `continue` 只会退出当前循环迭代的语句块。控制权又会回到 `while` 语句。如果 `while` 的测试条件为 `True`, 一个新的代码块迭代就会开始。

Python 中的下一个重要的循环构造是 `for` 循环。它看起来如下所示：

```
for item in <iterable>:
    code block
else:
    another code block
```

`for` 循环将各项从可迭代对象中取出, 并且对每一项都执行代码块。可以像之前在 `while` 循环中描述的那样使用 `break` 或者 `return` 来终止循环。可以像之前那样使用 `continue` 来终止循环中的单次迭代。

当所有迭代都执行完毕后, `else` 代码块就会被执行。如果使用 `break` 或者 `return` 退出循环, 它就不会被执行。

可迭代对象是任何遵守 Python 迭代器协议的对象。实际上, 它经常是一个容器, 比如一个列表、元组, 或者是一个可以返回一些值的函数, 比如 `range()`。`open` 函数返回一个文件迭代器。它有助于你在不需要先把文件读入内存的情况下循环遍历一个文件。也可以定义自己的自定义迭代类。

for 循环中有一个经常用到的函数是 `enumerate()`。这个函数返回包含迭代对象和一个序列数的元组。在默认情况下, 序列数等同于列表的下标。这意味着 `for` 代码块可以更加轻松地直接更新可迭代对象。`enumerate()` 接受第二个可选参数。这个参数指定了序列数的开始数字。例如, 可以使用这个参数来使文件中的行数从 1 而不是默认的 0 开始。

下面这个示例通过打印一个文件和关联的行数来说明这些知识点:

```
for number, line in enumerate(open('myfile.txt')):
    print(number, '\t', line)
```

最后, Python 有两个内嵌循环结构。你已经在本章早些时候列表的讨论中看到了它们中的一个: 列表推导。

列表推导是一种更一般的循环形式的特定应用。这种循环形式被称为生成器表达式。在有些地方, 如果不使用它, 可能就需要使用一个字面值的序列。如果回想一下本章早些时候列表推导的示例, 你使用生成器表达式把 1~10 的偶数的平方填充进了列表, 如下所示:

```
>>> [n*n for n in range(1,11) if not n*n % 2]
[4, 16, 36, 64, 100]
```

在方括号内的部分就是一个生成器表达式。它的一般形式如下:

```
<result expression> for <loop variable> in <iterable> if <filter expression>
```

通过比较这个一般式和列表推导的示例, 你会发现在示例中的结果表达式就是 `n*n`, 循环变量是 `n`, 可迭代对象是 `range(1,11)`。筛选表达式是 `if not n*n % 2`。

也可以像这样把它重写成常见的 `for` 循环:

```
result = []
for n in range(1,11):
    if n*n % 2:
        result.append(n)
```

关于生成器表达式, 有一点特别重要的是它们并不是一次就生成所有数据。更确切地说, 它们根据需要来生成(由此得名)数据项。这样当处理大型数据集时, 它可以极大地帮助节省内存资源。在本章后面, 你会在名为 `generator function` 的一种特殊的函数类型中学到更多关于生成器表达式的知识。

1.3.5 异常处理

有两种方法可以处理错误。第一种要在每个动作被执行时显式地检查它，另一种则尝试执行操作并且依赖系统在发生错误时产生一个错误条件或异常(exception)。尽管第一种方法在一些情况下非常适用，但在 Python 中，第二种方法则更为常用。Python 通过 try/except/else/finally 构造来支持这种技术。它的一般表达式看起来如下所示：

```
try:
    A block of application code
except <an error type> as <anExceptionObject>:
    A block of error handling code
else:
    Another block of application code
finally:
    A block of clean-up code
```

except、else 和 finally 都是可选的。但是如果使用 try 语句，则 except 和 finally 必须至少存在一个。结构中可以有多个 except 语句，但是只能有一个 else 或 finally 语句。如果不需要异常详情，则可以省略 except 语句行的 as...部分。

try 语句块被执行，如果发生了错误，就会测试异常类。如果存在与错误类型匹配的异常语句，就会执行相应的语句块(如果有多个异常语句块都指定同一个异常类型，则只有第一个匹配语句会被执行)。如果不存在匹配的 except 语句，异常会被向上传播一直到达顶层的解释器。Python 会产生其常规的错误追溯报告。注意，一个空的 except 语句可以捕捉任何错误类型。然而，这通常是一个糟糕的主意，因为它可能隐藏任何发生在非预期中的错误。

如果 try 代码块在没有任何错误的情况下执行成功，else 代码块就会执行。在实际中，else 很少被用到。不管是否有错误被捕捉或向上传播，也不管 else 语句是否被执行，finally 语句总是会被执行。这在锁定状态下提供了一个释放任何计算资源的机会。即使是在使用 break 或 return 语句退出 try/except 子句的情况下，finally 语句依然会被执行。

可以使用单一的 except 语句来处理多种异常类型。如果想要这样做，就要把异常类放在一个元组内(需要使用圆括号)。一个任意的异常对象包含异常发生位置的具体信息，同时提供了一个字符串转换方法。这样就可以通过打印这个对象来提供一个有意义的错误消息。

可以在你自己的代码中抛出异常，也可以使用任意现有的异常类型或通过创建一个 Exception 类的子类来定义你自己的异常类型。还可以给你抛出的异常传递参数，并且在 except 子句中使用错误对象的 args 特性来访问这些存在于异常对象中的参数。

下面这个示例抛出了带有一个自定义参数的 ValueError 异常，然后捕获这个错误并将给定的参数打印出来。

```
>>> try:
...     raise ValueError('wrong value')
... except ValueError as error:
...     print (error.args)
```

```
...
('wrong value',)
```

注意，你没有得到一个完整的回溯，只是打印了 `except` 代码块中的输出。也可以只是简单调用无参数的 `raise` 来处理它，然后重新抛出原异常。

1.3.6 上下文管理

Python 有一个运行时上下文(context)的概念。它通常包括一个临时性的资源。这个资源就是你的程序想要交互的一些东西。一个常见的示例可能是一个打开的文件或一个并发执行的线程。为了处理这个，Python 使用了关键字 `with` 和一个上下文管理器(context manager)协议。这个协议帮助你定义你自己的上下文管理器类，但是你在大部分情况下还是会使用 Python 提供的管理器。

你通过调用 `with` 语句来使用一个上下文管理器：

```
with open(filename, mode) as contextName:
    process file here
```

上下文管理器保证文件在使用后会被关闭。这对于上下文管理器来说是相当常用的功能。它可以保证宝贵的资源在使用后会被释放，或者对首次使用的资源采取适当的共享预防措施。上下文管理器通常可以避免使用 `try/finally` 结构。`contextlib` 模块为构建你自己的上下文管理器提供了支持。

现在你已经看到了 Python 可以处理的不同数据类型，以及在处理过程中可以使用的控制结构。现在是时候去探索如何在你的 Python 程序中读入和输出数据。这就是下一节的主题。

1.4 在 Python 中读取和输出数据

基本的数据输入和输出对于任何编程语言来说都是必要的。需要考虑你的程序会如何与用户和存储在文件中的数据进行交互。

1.4.1 与用户交互

如果想要通过 `stdout` 发送数据给用户，则可以使用已经多次见到的 `print()` 函数。在这部分，你会学到如何更加精确地控制输出。如果想要从用户读取数据，则可以使用 `input()` 函数。它会提示用户输入然后从 `stdin` 中返回原始字符的字符串。

相比第一次出现，`print()` 函数由于有几个可选参数要变得更加复杂些。最简单的用法是：你只是把一个字符串传给它，然后 `print()` 函数会在 `stdout` 上显示字符串，后面跟一个行结束符(eol)。稍微复杂点的用法是：可以一次性地传多个项给 `print()`，然后它会转换并且依次展示这些项。这些项之间用空格分隔。

上段文字在 `print()` 函数的行为中确定了三个固定元素：

- 它在 `stdout` 上展示输出。
- 它用一个 `eol` 符号结束。
- 它用空格分隔多个项。

实际上, 以上这些没有一个是固定的, `print()` 允许使用可选参数来修改任意或全部元素。可以通过指定一个 `file` 参数来修改输出; 分隔符是通过 `sep` 参数来定义的, 而结束字符是通过 `end` 参数定义的。下面这行打印了臭名昭著的拥有两个字符串的 "hello world" 信息, 用连字符(-)分隔, 以 "END" 作为结束标记, 并把它输出到文件中:

```
with open("tmp.txt", "w") as tmp:
    print("Hello", "World", end="END", sep="-", file=tmp)
```

这样, 文件的内容应该是: "Hello-WorldEND"。

字符串 `format()` 方法在与 `print()` 结合在一起时才会真正地体现它的作用。这个组合可以整洁清晰地分开呈现你的数据。此外, 在打印长列表的对象和字符串片段时, 使用 `format()` 通常会更加高效。Python 文档中有许多如何使用 `format()` 的示例。

也可以使用 `input()` 函数与用户进行交流。这个函数会读取用户对给定屏幕上的提示的响应值。需要自己负责把返回的字符转换成任何显示的数据类型, 并且要处理转换中出现的任何错误。



注意: 在 Python 版本 2 中, 使用的是 `raw_input()` 函数, 而不是 `input()`。在版本 2 中, `input()` 函数有非常不同的表现。它会对用户输入的任何东西都进行求值。这会产生一个安全问题, 因为用户可能输入恶意代码。版本 3 移除了版本 2 的 `input()` 函数, 然后把 `raw_input()` 重命名为 `input()`。

下面这个示例要求用户输入一个数字。如果数字太高或者太低, 它会打印一个警告(如果愿意的话, 它可以形成一个猜测游戏的核心)。

```
target = 66

while True :
    value = input("Enter an integer between 1 and 100")
    try:
        value = int(value)
        break
    except ValueError:
        print("I said enter an integer!")

if value > target:
    print (value, "is too high")
elif int(value) < target:
    print("too low")
```



```
else:
    print("Pefect")
```

程序提示用户输入一个适当范围内的整数,然后用 `int()` 把读取到的值转换成一个整数。如果转换失败,则会抛出一个 `ValueError` 异常,然后展示错误信息。如果转换成功,就肯定能得到一个有效的整数。这样,可以跳出 `while` 循环,继续把它和目标值进行测试。

1.4.2 使用文本文件

在编程中保存数据时,文本文件是非常重要的。Python 提供了几个函数用来处理文本文件。



注意: Python 中的文件接口实际上是一个更高层次抽象接口的具体化。抽象接口从一个名为 `io.IOBase` 的类开始。你基本上可以忽略它们。它们只是创建了标准的操作集,并把它们应用于文本文件和类文件对象。

在前面你看到了 `open()` 函数,它接受文件名和一个模式作为参数。模式可以为 `r`、`w`、`rw` 和 `a` 中的任意一个。它们分别代表着读、写、读写和附加(有一些其他模式,但是不经常用到。还有一些可选参数用来控制数据如何被解释。可以在文档中获得细节)。`r` 模式需要文件是存在的; `w` 和 `rw` 模式会创建一个新的空文件(或者覆盖任何已存在的同名文件)。`a` 模式会打开一个已经存在的文件。如果指定名称的文件并不存在,它会创建一个新的空文件。返回的文件对象也是一个上下文管理器。就像你在上下文管理器部分看到的那样,它可以被用在一个 `with` 语句块中。如果没有使用一个 `with` 语句块,那么当用完这个文件时,你应该调用 `close()` 方法显式地关闭文件。这样可以保证任何驻留在内存缓冲区中的数据都会被发送到磁盘上的物理文件中。`with` 结构体会自动地调用 `close()`,这也是使用上下文管理器方法的优势之一。

一旦已经打开了文件对象,就可以根据需要使用 `read()`、`readlines()` 或 `readline()` 这些方法。`read()` 方法读取整个文件内容到一个字符串中,并以一个换行符结束。`readlines()` 方法把文件一行一行地读入一个列表中,并且保留每一行的换行符。`readline()` 方法读取文件中的下一行,它也保留换行符。文件对象是可迭代的,所以你不需要任何读取方法就可以直接在 `for` 循环中使用它。因此,从文件中读取行的推荐模式如下:

```
with open(filename, mode) as fileobject:
    for line in fileobject:
        # process line
```

可以使用 `write()` 或 `writelines()` 方法向一个可写入的文件对象中写入数据。它们与类似名称的读方法是等价的。注意,没有写单独一行的 `writeline()` 方法。

如果在使用 `rw` 模式,你可能想要移动到文件的特定位置来重写现有的数据。使用 `tell()`

方法可以找到你在文件中的当前位置。使用 `seek()` 方法可以定位到一个指定位置(可能是你之前使用 `tell()` 记录下来的一個位置)。`seek()` 有几种计算位置的模式, 默认模式是计算距离文件开头的偏移值。

现在你已经掌握了编写 Python 程序的所有的基本技术。然而, 为了实现本书的重点, 即能够处理更大的项目, 你会想要扩展 Python 的功能。下一节我们开始探索它。

1.5 扩展 Python

最简单的扩展 Python 的方法就是编写你自己的函数。你定义的函数可以与使用它们的代码放在相同的文件中。或者也可以创建一个新的模块, 然后从它导入函数。你会在下一节看到模块。现在, 你将会创建函数并且在同一个文件中使用它们。实际上在这一节, 你大多数时间会使用交互式提示符来试验示例。

Python 创建新功能中的下一步就是定义你自己的类并且从它们创建对象。同样, 普遍的做法是在模块中创建类。你会在下一节看到怎么实现它。这部分涉及的示例都很简单, 你仅仅使用 Python 提示符就可以了。

Python 程序员经常在他们的程序中使用文档字符串。文档字符串就是字符串面值。它们没有被赋值给变量, 而且遵循在它们被定义地方的缩进级别。可以用文档字符串来描述函数、类或者模块。`help()` 函数会读取并展示这些文档字符串。

1.5.1 定义并使用函数

Python 中有多种类型的函数。这部分会首先介绍标准函数, 然后是生成器函数, 最后是有些神秘的 `lambda` 函数。

在 Python 中, 你会使用 `def` 关键字来定义函数。它看起来如下所示:

```
def functionName(parameter1, param2,...):  
    function block
```

Python 函数永远会返回一个值。可以用 `return` 关键字来指定一个显式的返回值。否则, Python 会默认返回 `None`(如果在输出中发现意想不到的 `None` 值, 请检查相关函数体中是否含有显式的 `return` 语句)。可以通过在参数名后面加上等号和指定值来给参数赋予默认值。在下面, 你会在 `odds()` 生成器函数中看到一个示例。

通过下面的“试一试”中, 可以最容易理解一个函数定义是怎么创建和使用的。

试一试: 创建并使用一个函数

在这个“试一试”中, 你会创建一个函数。这个函数接受多个输入参数并返回一个值。此函数会根据给定的斜率、 x 坐标和常量, 使用直线的数学方程, 返回对应的 y 坐标。然后, 你会使用这个函数去生成一条直线上的一系列坐标。

(1) 启动 Python 解释器。

(2) 输入下面的代码来定义函数：

```
>>> def straight_line(gradient, x, constant):
...     ''' returns y coordinate of a straight line
...         -> gradient * x + constant'''
...     return gradient*x + constant
...
>>>
```

(3) 既然你已经定义了函数，使用一些简单的值来测试它。可以提前心算出这些值。试着使用斜率值为 2、x 值为 4、常数为 -3 的参数来调用函数：

```
>>> # test with a single value first
>>> straight_line(2,4, -3)
5
```

(4) 现在让我们用一种更加复杂的方法来测试函数。使用下面的代码：

```
>>> for x in range(10):
...     print(x, straight_line(2,x,-3))
...
0 -3
1 -1
2 1
3 3
4 5
5 7
6 9
7 11
8 13
9 15
```

(5) 最后，检查 help() 函数能否正确地识别函数：

```
>>> help(straight_line)
Help on function straight_line in module __main__:

straight_line(gradient, x, constant)
    returns y coordinate of a straight line
    -> gradient * x + constant
(END)
```

示例说明

第二步的第一行创建了函数定义。函数被命名为 `straight_line`，并且有三个必要的函数： `gradient`、`x` 和 `constant`。这些参数对应着在数学方程式 $y=mx+c$ 中使用的值。在这个方程式中， m 是斜率， c 是常量。

第二行是一个文档字符串。它描述了这个函数的功能以及应该如何使用它。

第三行是函数的代码块。它可能任意的复杂并且长达多行。但是在这个示例中，它只

有一行。因为要返回结果，所以在前面加了关键字 `return`。注意，代码行首的缩进级别应与文档字符串的行首相同。否则，你会得到一个缩进错误。

然后，我们使用一些简单值来测试函数。通过心算，我们确定返回值 5 确实等于 $(2*4-3)$ 。看起来，函数至少在一些简单的情况下能够正常工作。

我们在 `for` 循环中使用函数来产生一组 x, y 坐标对。在这里，`gradient` 的值被固定为 2，`constant` 值被固定为 -3，而把 x 作为循环变量。如果身边有便利的纸的话，可以尝试在纸上画出函数产生的坐标，并确认这些坐标能否形成一条直线。

最后，我们使用 `help()` 函数来确定文档字符串被正确地检测到并展示出来。

1. 生成器函数

你将看到的下一个函数形式是生成器函数。生成器函数与标准函数看起来几乎一样，除了标准函数使用 `return` 返回数据，而生成器函数使用关键字 `yield` (理论上生成器函数除了使用 `yield` 之外也可以使用 `return`，但是只有 `yield` 表达式能产生生成器行为)。

Python 优雅的魔力使生成器函数很特殊。它们像定格相机一样工作。当一个标准函数遇到 `return` 语句时，它会返回值，然后函数就会丢弃它的所有内部数据。当下次函数被调用时，一切都从头做起。

`yield` 语句会做不同的事情。它像 `return` 一样返回一个值，但是它不会使函数丢弃数据；相反，所有数据都被保存起来了。下次函数被调用时，即使 `yield` 语句在代码块的中间或者处于循环中，程序也会从 `yield` 语句开始执行。在一个函数中，甚至可以有多个 `yield` 语句。由于 `yield` 语句可以被放在一个循环中，这就可以创建一个高效的返回一个无限系列结果的函数。下面的示例返回一个递增的奇数序列：

```
def odds(start=1):
    ''' return all odd numbers from start upwards'''
    if int(start) % 2 == 0: start = int(start) + 1
    while True:
        yield start
        start += 2
```

在这个函数中，首先检查 `start` 参数是否是一个奇数(偶数除以 2 的余数是 0)。如果不是奇数，你会强迫它加 1 而成为下一个最近的奇数。然后，你创建了一个无限的 `while` 循环。通常，这是一个糟糕的主意，因为你的程序会陷入死循环。然而，由于这是一个生成器函数，你在使用 `yield` 语句返回 `start` 的值。这样，函数在返回 `start` 的值时就会退出。当下一次函数被调用时，程序会从前次离开的地方再开始。所以 `start` 会被加 2，然后继续循环，产生下一个奇数并且退出函数。函数会在下一次调用时继续。

Python 确保生成器函数能够变成迭代器，这样你就可以在 `for` 循环中使用它们。如下所示：

```
for n in odds():
    if n > 7: break
```

```
else: print(n)
```

你把 `odds()` 当成一个集合在使用。每次循环访问它，它会调用生成器函数，然后接收下一个奇数值。

通过插入 `break` 测试，可以避免一个无限循环。这样，当大于 7 时，`odds()` 就不会被调用了。



注意：如果在同一个程序中第二次使用 `odds()`，它会创建一个全新的迭代器实例，并且序列会从头开始。

现在你已经理解生成器函数是如何工作了，你也可能已经认识到本章前面介绍的生成器表达式就是高效的匿名生成器函数。生成器表达式实际上是一种变相的没有名称的生成器函数。

这为我们将要学到的最后一个函数类型，`lambda` 函数，提供了一个完美的桥梁。

2. `lambda` 函数

术语 `lambda` 来自于阿隆佐·邱奇(Alonzo Church)发明的一种微积分。好消息是，你不需要知道任何使用 `lambda` 函数的数学知识。`lambda` 函数背后的原理是它通常是一个很小的匿名函数块。可以把它插入到代码中，然后像一个普通函数一样调用 `lambda` 函数。`lambda` 函数并不常用。但是当需要创建很多只会被使用一次的小函数时，它们是非常方便的。它们通常被用在 GUI 或网络编程的环境中。在这些情况下，编程工具包需要一个用来回调得到结果的函数。

`lambda` 函数的定义如下所示：

```
lambda <param1, param2, ..., paramN> : <expression>
```

这是一个 `lambda` 字面值加上一个可选的逗号分隔的参数名列表、一个冒号和一个任意的 Python 表达式。这个表达式通常会使用输入参数。注意，表达式前面并没有关键字 `return`。

一些语言允许 `lambda` 函数可以任意的复杂。但是 Python 只允许使用一个表达式。这个表达式也可以非常复杂。但是在实际中，这更适合创建一个标准的函数。这样代码具有更好的可读性，在出现问题时，也更便于调试。

可以把 `lambda` 函数赋值给一个变量。这种情况下，那些变量看起来就像是标准的 Python 函数名。例如，以下是一个用 `lambda` 函数重新实现的 `straight_line` 函数的示例：

```
>>> straight_line = lambda m,x,c: m*x+c
>>> straight_line(2,4, -3)
5
```

在本书之后的部分中，你会经常看到 `lambda` 函数的身影。切记：它们就是用来简明地表达那种简短的单行表达式的函数。

1.5.2 定义并使用类和对象

Python 使用一种传统的、基于类的方法支持面向对象编程。Python 类支持多继承和操作符重载(但不支持方法重载), 以及常用的封装机制和消息传递。尽管一些命名习惯可以为特性提供一层微弱的保护并建议客户端不应该在何时直接使用特性, 但是 Python 类不直接实现数据隐藏。Python 支持类方法和类数据, 以及属性(properties)和插槽(slots)的概念。类拥有构造函数(`__new__()`)和初始化函数(`__init__()`)。尽管并不保证被调用, Python 也有析构函数机制(`__del__()`)。对于定义在类内部的方法和数据, 类也充当它们的命名空间。

对象是类的实例。尽管不常用, 但是实例在创建之后可以添加自己的属性。

类的定义使用 `class` 关键字, 后面跟着类名和用括号括起来的超类的列表。类定义包含一些类数据和方法定义。一个类方法定义把指向调用实例的引用作为第一个参数, 通常被称为 `self`。一个简单的类定义看起来如下所示:

```
class MyClass(object):
    instance_count = 0
    def __init__(self, value):
        self.__value = value
        MyClass.instance_count += 1
        print("instance No {} created".format(MyClass.instance_count))
    def aMethod(self, aValue):
        self.__value *= aValue
    def __str__(self):
        return "A MyClass instance with value: " + str(self.__value)
    def __del__(self):
        MyClass.instance_count -= 1
```

类名通常以一个大写字母开头。在 Python 3 中, 除非特别声明, 超类总是 `object`。所以前面示例使用 `object` 作为超类实际上是多余的。由于没有出现在任何类方法中, 因此 `instance_count` 数据项是一个类特性。`__init__()` 函数是一个初始化器(在 Python 中, 除非是从一个内置类继承, 构造函数很少被用到)。它设置了实例变量 `self.__value` 的值, 添加之前定义的类变量 `instance_count`, 然后打印一条信息。`value` 前的双下划线表明它实际上是一个私有数据, 不应该被直接使用。在对象被构建之后, `__init__()` 方法立刻被 Python 自动调用。实例方法 `aMethod()` 修改在 `__init__()` 方法中创建的实例特性。`__str__()` 方法是一个用来返回一个格式化字符串的特殊方法。例如, 传递到打印函数的对象会以一种有意义的方法被打印出来。当对象被销毁时, 析构函数 `__del__()` 减少类变量 `instance_count`。

可以如下所示创建一个类的实例:

```
myInstance = MyClass(42)
```

该操作在内存中创建了一个实例, 然后把新实例作为 `self`, 42 作为 `value` 来调用 `MyClass.__init__()`。

可以如下所示使用点符号调用 `aMethod()` 方法:

```
myInstance.aMethod(66)
```


这可以被转换成更加显式的调用：

```
MyClass.aMethod(myInstance, 66)
```

并且产生想要的行为。这样，`__value` 特性的值被调整了。

如果打印实例，则可以看到 `__str__()` 方法起的作用：

```
print(myInstance)
```

该操作会打印下面这条信息：

```
A MyClass instance with value: 2772
```

也可以在创建或销毁一个实例之前和之后打印 `instance_count` 的值：

```
print(MyClass.instance_count)
inst = MyClass(44)
print(MyClass.instance_count)
del(inst)
print(MyClass.instance_count)
```

这会显示计数被增加然后又被减少(在垃圾回收过程中，在析构器调用之前可能会有一点小小的延迟，这应该只是一小会儿)。

`__init__()`、`__del__()` 和 `__str__()` 方法不是仅有的特殊方法。它们中的一些都使用双下划线来标识(它们有时也被称为双下划线(dunder)方法)。操作符重载是通过一组这些特殊的方法来支持的，包括 `__add__()`、`__sub__()`、`__mul__()` 和 `__div__()` 等。其他方法为实现 Python 协议，比如迭代或上下文管理，做了准备。可以在自己的类中重写这些方法。你永远都不要定义自己的以双下划线开始的函数，否则 Python 未来的改进和增强可能会破坏你的代码。

可以在子类中重写方法，并且新的定义可以通过使用 `super()` 函数触发被继承版本的方法。如下所示：

```
class SubClass(Parent):
    def __init__(self, aValue):
        super().__init__(aValue)
```

对 `super().__init__()` 的调用会转换成对父类的 `__init__()` 方法的调用。使用 `super()` 可以避免问题，尤其是在多继承的情况下。在多继承的情况下，一个类可能被继承多次，但是通常你不想它被多次初始化。



注意：相比于 Python 2，在 Python 3 中使用 `super()` 被大大简化了。Python 2 中的 `super()` 看起来就像 `super(SubClass, self).__init__(aValue)`，但这种方式用起来非常不直观。

插槽是一种节省内存的设备。可以使用 `__slots__` 这个特殊特性并提供一个对象特性名

列表来激活它们。通常，`__slots__` 是一个不成熟的优化。只有当有一个明确的、已知的需求时，才应该使用它。

属性是另一个数据特性中可用的特性。即使没有使用常用的方法语法，它们也会强制通过一组方法访问特性。这样就允许你让这个特性只读(或者甚至是只写)。可以通过一个示例透彻地理解它。这个示例创建一个 `Circle` 类，包括 `radius` 特性和 `area()` 方法。由于你希望 `radius` 的值永远为正，因此你不希望用户可以直接改变它的值，因为它们可能会传一个负值。即使 `area()` 被实现为一个方法，你也希望它看起来像一个只读的数据特性。可以通过把它作为 `radius` 和 `area` 属性来同时达到这两个目的。

试一试：在类中创建属性(testCircle.py)

在这个“试一试”中，首先会创建一个简单的 `Circle1` 类。它只有一个特性和两个可调用的方法：`setRadius()` 和 `area()`。然后，会创建第二个类 `Circle2`。它把 `radius` 和 `area` 变成属性。最后，你会看到如何使用属性来简化使用客户端代码中的类。

(1) 启动你最喜欢的编程编辑器或 IDE，创建一个名为 `testCircle.py` 的新文件(或者加载从本书网站上下载的文件)。

(2) 输入下面的代码：

```
class Circle1:
    def __init__(self, radius):
        self.__radius = radius
    def setRadius(self, newRadius):
        if newRadius >= 0:
            self.__radius = newRadius
        else: raise ValueError("Value must be positive")
    def area(self):
        return 3.14159 * (self.__radius ** 2)

class Circle2:
    def __init__(self, radius):
        self.__radius = radius

    def __setRadius(self, newRadius):
        if newRadius >= 0:
            self.__radius = newRadius
        else: raise ValueError("Value must be positive")
    radius = property(None, __setRadius)

    @property
    def area(self):
        return 3.14159 * (self.__radius ** 2)
```

(3) 保存代码。

(4) 启动 Python 解释器，输入下面代码来使用 `Circle1`：

```
>>> import testCircle as tc
```

```
>>> c1 = tc.Circle1(42)
>>> c1.area()
5541.76476
>>> print(c1.__radius)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: 'Circle1' object has no attribute '__radius'
>>> c1.setRadius(66)
>>> c1.area()
13684.766039999999
>>> c1.setRadius(-4)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "D:\PythonCode\Chapter1\testCircle.py", line 7, in setRadius
    else: raise ValueError("Value must be positive")
ValueError: Value must be positive
```

(5) 用下面的代码来试试 Circle2:

```
>>> c2 = Circle2(42)
>>> c2.area
5541.76476
>>> print(c2.radius)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
AttributeError: unreadable attribute
>>> c2.radius = 12
>>> c2.area
452.38896
>>> c2.radius = -4
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "D:\PythonCode\Chapter1\testCircle.py", line 18, in __setRadius
    else: raise ValueError("Value must be positive")
ValueError: Value must be positive
>>>
```

示例说明

在文件 `testCircle.py` 中创建了两个类。第一个类 `Circle1` 实现了你想要强制用户使用 `setRadius()` 方法修改半径值的目的。通过在特性 `self.__radius` 前面加上双下划线前缀实现了这一点。这是 Python 所用的私有化方法。然后你创建了 `setRadius()` 方法，它会在使用给定值之前检验它的有效性。如果值是负的，它会抛出一个错误。你也提供了一个 `area()` 方法，这样用户可以使用通常的方法调用技术来计算面积。

第二个类 `Circle2` 使用相当不同的方式处理这些需求。它使用 Python 中的属性定义特性创建了一个只写特性 `radius`。它也创建了一个作为只读特性的 `area` 方法。就像在解释器中练习使用类看到的那样，用户在使用 `Circle2` 时的代码更加直观。这关键在于你调用的 `property()` 类型函数，如下所示：


```
radius = property(None, __setRadius)
```

这行代码接受的参数是一组函数，它们用来执行读、写和删除操作(以及文档字符串)。每个函数的默认值都是 `None`。这里创建了 `radius` 属性，设置读函数为 `None`，但是写函数为 `__setRadius()` 方法(现在是私有的)。其他参数值都被默认地设置为 `None`。这样做的结果是：用户可以把它当成一个公共的数据特性访问，但是当给 `radius` 赋值时，就必须调用 `__setRadius()` 方法来保护这个值。任何试图读取(或删除)这个特性的操作都会被忽略，因为这些操作实际上操作的都是 `None`。

`area` 属性稍有差别，它使用了一个 Python 属性修饰符(`@property`)。属性修饰符是一种创建只读属性的快捷方式。这是属性的一种常用方式。

下面介绍交互式会话，我们创建了一个 `Circle1` 实例，并且使用 `area()` 方法打印出面积。然后试图通过访问 `__radius` 直接打印半径，但是 Python 假称它没有这个特性(由于双下划线的私有设置)并且抛出一个 `AttributeError` 异常。但是当使用 `setRadius()` 方法时，一切就很顺利。再次打印面积时，修改已经生效了。最后，我们试图把一个负值赋给半径，正如我们所料，`setRadius()` 方法抛出了一个 `ValueError` 异常。这个异常还带有一个自定义的错误消息：“Value must be positive”。

在使用 `Circle2` 的会话中，我们可以看到代码变得更加简洁。我们仅使用 `area` 属性名就可以获得面积，并且也可以给 `radius` 属性名赋值。当试图将一个负值赋给它时，赋值方法抛出了一个 `ValueError` 异常。尽管错误消息会稍有不同，但直接打印半径又产生了一个 `AttributeError` 异常。

属性需要程序员这边做少量的额外工作，但却可以极大地简化类的使用。

在这一节中，我们已经看到了如何使用函数和类来扩展 Python 的功能。在下一节中，你将看到如何把这些扩展封装在可复用的模块和包中。

1.6 创建和使用模块和包

在大多数编程环境中，模块对于正式的编程任务来说是非常重要的。它们允许程序被拆分为易于管理的块并且提供项目间代码复用的机制。在 Python 中，模块只是简单的以 `.py` 结尾的源文件，它们位于任何 Python 可以找到的地方。在实际中，这意味着文件必须位于当前的工作目录或在 `sys.path` 变量列出的文件夹中。通过在系统的环境变量 `PYTHONPATH` 中指定你自己的文件夹或在运行时动态地添加，可以把它们添加到这个路径中。

尽管模块提供了一个很有用的包装方法来封装小数量的源文件用于复用，但是它们并不能完全满足更大的项目，例如，GUI 框架或数学函数库。对于这些，Python 提供了包(package)的概念。一个包本质上是包含许多模块的文件夹。唯一的要求是，文件夹需要包含一个名为 `__init__.py` 的文件，这个文件可以为空。对于用户来说，包看起来特别类似于模块，包中的子模块看起来就像模块的特性。

1.6.1 使用和创建模块

我们使用 `import` 关键字来访问模块。在 Python 中，它有很多变形，最常见的形式如下：

```
import aModule
import aModule as anAlias
import firstModule, secondModule, thirdModule...
from aModule import anObject
from aModule import anObject as anAlias
from aModule import firstObject, secondObject, thirdObject...
from aModule import *
```

最后一种形式把 `aModule` 模块中所有可见的名称都导入到当前的命名空间中(很快你会学到如何控制可见性)。该操作可能存在命名冲突的严重风险，导入的名称可能会与内置名称或者你已经或将要在本地定义的名称相冲突。因此，最后一种导入形式只推荐在 Python 命令提示符中测试模块时使用。相比于命名冲突带来的混乱来说，使用其他导入形式带来的小量额外输入的代价其实是很小。其他 `from...` 形式要更加安全，因为你只是导入指定的名称，而且在必要时还可以使用别名来重命名它们。这样可以大大降低与本地名称发生命名冲突的概率。

一旦使用前三种形式中的任意一种来导入模块，就可以通过在需要的名称前加上模块名(或别名)和点符号来访问它的内容。在前面，你已经学习了有关这方面的示例。例如，`sys.path` 就是 `sys` 模块的一个特性。

既然已经介绍了模块实际上仅仅是源文件，那么你就应该了解一下在创建模块时的一些注意事项。也许除了初始化那些可能依赖本地环境的变量外，你应该避免使用顶层代码，它们会在模块被导入时运行。这意味着你想要复用的代码应该被包装成函数或者类。通常，也会提供测试函数来练习使用模块中的所有函数和类。模块的名称习惯上也都是小写字母。



注意：有一个称为 PEP8 的 Python 风格指南，它提供了一些命名习惯和代码布局规则方面的指导。这些指导并不是强制的，但是强烈推荐你使用它，尤其在你提交的代码被包含在标准库时。PEP8 可以在下面网站中找到：<http://legacy.python.org/dev/peps/pep-0008/>。

有两种方法可以控制模块中对象的可见性。第一种方法类似于类中的私有机制，可以在一个名称前面加一个下划线前缀。当使用 `from x import *` 风格的语句时，这些名称并不会被导出。另一种控制可见性的方法就是在一个名为 `__all__` 的顶层变量中列出你想要导出的名称。这种方法可以保证只有那些你特别想要被导出的名称才会被导出。如果可见性对你来说很重要的话，我们推荐你使用这种方法，而不是下划线方法。



注意：模块中一个重要的陷阱是 `sys.path` 列表是按序搜索的。这通常意味着，你创建的任何模块都会在内置或标准库模块之前被找到。所以你不要使用一个标准模块名作为你自己模块文件的名称，这一点很重要。否则，会发生奇怪的事情，并且即使你意识到那是你的模块，其他读者读取它时也很容易被捉弄。

在下一个“试一试”中，你会练习大部分这些内容，但是首先需要了解一下包以及它们和模块之间的区别。

1.6.2 使用和创建包

在本节的开始，你发现一个 Python 包其实仅仅是包含一个名为 `__init__.py` 文件的文件夹。所有在这个文件夹中的其他 Python 文件都是这个包的模块。Python 把包仅看成另一个类型的模块。这意味着 Python 包可以包含其他包到任意深度。只要每个子包都有自己的 `__init__.py` 文件，它就是个有效的包。



注意：刚才说包被定义为拥有一个 `__init__.py` 文件，但严格说来，这不是真的。包的真正定义特征是它有一个 `__path__` 特性。然而，在实际中，你不需要提供它，因为 Python 会为你完成这件事。所以，如果创建一个 `__init__.py` 文件，一切就都就绪了。

`__init__.py` 文件本身并不是非常特殊，它只是另一个 Python 文件。当导入包时，它就会被加载。这意味着这个文件可以为空，这种情况下，导入包的操作仅仅是让你能够访问包里的模块。`__init__.py` 文件也可以像任何其他模块一样包含 Python 代码。特别是，它可以像之前描述的那样定义一个 `__all__` 列表来控制可见性，这可以有效地让包拥有私有的实现文件。当一个客户导入包时，这些文件不会被导出。

在创建包时，一个常见的需求是需要有一组函数或数据，能够在所有被包含的模块之间分享。可以把被分享的代码放入一个名为 `common.py` 的模块文件中并把它放在包的顶层，然后让所有其他模块都 `import common`，这样就能满足这个需求。它会作为包的一部分而对它们可见，但如果不是显式地被包含在 `__all__` 列表中，包的外部消费者不会看到它。

当使用包时，可以像对待任何其他模块一样对待它。你使用所有常用的风格来导入变量。但是命名模式被扩展了，需要使用点符号来指定在包的层级中哪个子模块是需要的。当使用点符号导入子模块时，实际上定义了两个新的名称：包名和模块名。例如，考虑一下这条语句：


```
import os.path
```

这行代码导入了 `os` 包的 `path` 子模块。但是它同时也让 `os` 模块作为一个整体可见。你不需要再为 `os` 本身使用单独的导入语句就可以继续访问 `os` 的函数。可以推断：Python 自动执行了在导入层级中的所有 `__init__.py` 文件。所以在 `os.path` 的示例中，它先执行了 `os.__init__.py` 文件，然后执行了 `path.__init__.py` 文件。

另一方面，如果按如下方式在导入之后使用别名：

```
import os.path as pth
```

则只有 `os.path` 模块是被暴露的。如果想使用 `os` 模块函数，就需要一个显式的导入。尽管只有 `path` 作为名称 `pth` 是被暴露的，但 `os` 和 `path` 的 `__init__.py` 文件还是都被执行了。

Python 标准库包含很多个包，其中就包含了刚才提到的 `os` 包。其他值得注意的包包括 UI 框架 `tkinter` 和 `curses`、`email` 包，以及关注于网络的 `urllib`、`http` 和 `html` 包。你将会在本书的后面部分使用它们中的一些。

命名空间包：

Python 3.3 引入了一种新类型的包，称为命名空间包(namespace package)。一个命名空间包包含数个部分。每一部分都指向一个对象。这个对象有或者没有一个物理的表示形式，并且可能位于网络上或者在本地文件系统的不同部分中。命名空间包并不使用 `__init__.py` 文件技术，而是依赖于 `sys.path` 的定义。它会在导入时寻找模块。

在撰写本书时命名空间包的版本还很新，它们有什么其他的扩展用法还未可知。就眼前来看，你可能在实际中不会遇到它们。而这意味着对于用户而言，它们看起来与传统的包区别不会很大。

现在你已经了解了所有关于模块和包的原理。在下一节中，我们会创建一些模块和一个包。这会帮助你熟悉这些原理。

1.7 创建示例包

既然你已经了解了相关原理知识，是时候把它们付诸实践。在本节中，将创建两个模块，并且把它们绑定在同一个包中。你会使用在布尔类型部分提到的按位逻辑操作符。包的目的是为那些操作符提供一个基础的接口，并且扩展它们的应用范围，以包括测试单个的位值。在实现这一点时，你也会看到几个之前介绍过的 Python 的核心语言特性。你开发的模块并不是用来优化性能，而是被设计来说明概念。然而，把它们完善成真正有用的工具并非难事。

试一试：创建模块(bits.py)

在这个“试一试”中，首先会创建一个简单的、传统的基于整数输入的模式。然后，会创建另一个模块。它定义了一个可以用来表示一块二进制数据的类。这个类提供了一些

按位函数作为类方法。最后，创建了一个包来包含这两个模块。

(1) 新建一个名为 **bitwise** 的文件夹。它最终会成为你的包。

(2) 在这个文件夹中，创建一个名为 **bits.py** 的 Python 脚本文件。这个文件包含下面的代码(或者加载本书的下载文件中名为 **bits.py** 的文件):

```
#!/bin/env python3
''' Functional wrapper around the bitwise operators.
Designed to make their use more intuitive to users not
familiar with the underlying C operators.
Extends the functionality with bitmask read/set operations.
```

```
The inputs are integer values and
return types are 16 bit integers or boolean.
bit indexes are zero based
```

Functions implemented are:

```
NOT(int)          -> int
AND(int, int)     -> int
OR(int,int)       -> int
XOR(int, int)     -> int
shiftright(int, num) -> int
shiftright(int, num) -> int
bit(int,index)    -> bool
setbit(int, index) -> int
zerobit(int,index) -> int
listbits(int,num) -> [int,int...,int]
```

```
...
```

```
def NOT(value):
    return ~value
```

```
def AND(val1,val2):
    return val1 & val2
```

```
def OR(val1, val2):
    return val1 | val2
```

```
def XOR(val1,val2):
    return val1^val2
```

```
def shiftright(val, num):
    return val << num
```

```
def shiftright(val, num):
    return val >> num
```

```
def bit(val,idx):
```

```

mask = 1 << idx # all 0 except idx
return bool(val & 1)

def setbit(val, idx):
    mask = 1 << idx # all 0 except idx
    return val | mask

def zerobit(val, idx):
    mask = ~(1 << idx) # all 1 except idx
    return val & mask

def listbits(val):
    num = len(bin(val)) - 2
    result = []
    for n in range(num):
        result.append( 1 if bit(val, n) else 0 )
    return list( reversed(result) )

```

(3) 保存文件，就在你的 bitwise 文件夹中，启动 Python 解释器。

(4) 输入下面的代码，测试你的新模块：

```

>>> import bits
>>> bits.NOT(0b0101)
-6
>>> bin(bits.NOT(0b0101))
'-0b110'
>>> bin(bits.NOT(0b0101) & 0xF)
'0b1010'
>>> bin(bits.AND(0b0101, 0b0011) & 0xF)
'0b1'
>>> bin(bits.AND(0b0101, 0b0100) & 0xF)
'0b100'
>>> bin(bits.OR(0b0101, 0b0100) & 0xF)
'0b101'
>>> bin(bits.OR(0b0101, 0b0011) & 0xF)
'0b111'
>>> bin(bits.XOR(0b0101, 0b11) & 0xF)
'0b110'
>>> bin(bits.XOR(0b0101, 0b0101) & 0xF)
'0b0'
>>> bin(bits.shiftright(0b10, 1))
'0b100'
>>> bin(bits.shiftright(0b10, 4))
'0b100000'
>>> bin(bits.shiftright(0b1000, 2))
'0b10'
>>> bin(bits.shiftright(0b1000, 6))
'0b0'
>>> bits.bit(0b0101, 0)
True

```



```
>>> bits.bit(0b0101,1)
False
>>> bin(bits.setbit(0b1000,1))
'0b1010'
>>> bin(bits.zerobit(0b1000,1))
'0b1000'
>>> bits.listbits(0b10111)
[1, 0, 1, 1, 1]
```

示例说明

这个模块是一个非常简单的函数列表。这些函数包装了内置的按位操作符，包括非(~)、与(&)、或(|)、异或(^)、左移(<<)和右移(>>)。这些操作作用于二进制数据，也就是简单的由 1 和 0 组成的、并且在计算机中作为一个存储单元被存储起来的一个序列。基本上，所有数据都是以二进制形式被存储在计算机中的。

基于这些包装函数，又补充了一组函数。它们的功能分别是测试某个位的值是否是 1，设置一位的值为 1，以及设置一位的值为 0(或者称为重置位值)。位数是从右边开始查的，起始于 0。测试是通过使用一个位模式(或者称为位掩码(bitmask))完成的。在除了 zerobit() 之后的所有情况中，这个位模式就是除了你想要测试或设置的位之外，其他位都是 0 的序列。可以通过将 1 左移一定的数位来创建掩码。zerobit() 对普通的掩码进行按位补码，这样可以创建一个除了你想要重设的位为 0，其他位都是 1 的序列。

最后，使用一个函数列出给定值的每一位。最后一个函数稍微有一些复杂，它体现了一些 Python 的编程特色。首先，通过 bin() 把它转换成一个二进制字符串，并把这个字符串的长度减 2(为了去掉开头的 0b 字符)来计算出数字的长度。然后，创建了一个空的结果列表，并且在得到的位组上进行循环。对于每一位，你会根据该位是否被设置来向列表中添加 1 或 0，这使用了 Python 的条件表达式结构。

对模块的测试引入了一些有趣的话题。最开始，导入了创建的新模块。由于当前位置与文件在同一个文件夹中，所以不需要修改 sys.path 的值 Python 就可以查找到它。在测试的开始，测试了 NOT() 函数(当然，前面加了模块名 bits 作为前缀)，然后马上就能看到一反常现象：Python 解释器打印了十进制的表达式作为结果。为了解决这个问题，可以使用 bin() 函数将数字转换成一个二进制的字符串表达式。然而，还有个问题，因为数字是负的。Python 在内部会使用最左边的位来表示符号。通过反转所有位，我们也可以反转它的符号。可以使用一个掩码 0xF(或者十进制 15，如果喜欢的话)来避免这种混淆。这个掩码帮助只恢复最右的四位。可以使用 bin() 来进行这个转换，你现在可以看到预期中的反转位模式。很明显，如果反转的值大于 16，你可能需要使用更长的掩码。只需要记住每个十六进制数都是 4 位。所以需要做的就是掩码后面额外加一个 F。

下一组测试非常简单，测试的函数范围从 AND() 到 shiftright()。可以在视觉上检查输入位模式和输出检查结果。shiftright() 的示例展示了一个有趣的输出，它把位向右移了太多，以至于产生了一个 0 的结果。换言之，Python 使用 0 去填充移位操作留下的空隙。

下面介绍下一个新功能，你使用了 bit()、setbit() 和 zerobit() 来测试和修改给定值的单

独位。同样，可从视觉上检查输入和输出的模式来确定它们是否产生了正确的结果。记住，索引参数从右边开始记数并且起始于 0。

最后，测试了 `listbits()` 函数。再一次，可以轻松地对比二进制输入模式和数的结果列表。

所以，现在你已经有了一个可以被导入的工作模块。可以像 Python 中任何其他模块一样使用它。可以通过提供一个测试函数来进一步增强模块。并且如果愿意，还可以把这个测试函数包装在一个 `if __name__` 子句中。但是现在，你将继续了解如何从一个单独的模块开发成一个包。

试一试：创建包(bitmask.py)

在这个“试一试”中，构建了一个类。它复制了所有在 `bits.py` 中的函数作为一组方法。然后将两个模块都绑定到一个包中。

(1) 进入 `bitwise` 文件夹。

(2) 创建一个名为 `bitmask.py` 的新文件。这个文件中包含下面的代码(或者加载本书的下载文件中名为 `bitmask.py` 的文件)：

```
#!/bin/env python3
''' Class that represents a bit mask.
It has methods representing all
the bitwise operations plus some
additional features. The methods
return a new BitMask object or
a boolean result. See the bits
module for more on the operations
provided.
'''
```

```
class BitMask(int):
    def AND(self, bm):
        return BitMask(self & bm)
    def OR(self, bm):
        return BitMask(self | bm)
    def XOR(self, bm):
        return BitMask(self ^ bm)
    def NOT(self):
        return BitMask(~self)
    def shiftleft(self, num):
        return BitMask(self << num)
    def shiftright(self, num):
        return BitMask(self > num)
    def bit(self, num):
        mask = 1 << num
        return bool(self & mask)
    def setbit(self, num):
        mask = 1 << num
        return BitMask(self | mask)
```

```

def zerobit(self, num):
    mask = ~(1 << num)
    return BitMask(self & mask)
def listbits(self, start=0, end=-1):
    end = end if end < 0 else end+2
    return [int(c) for c in bin(self)[start+2:end]]

```

(3) 现在保存它，可以在 Python 解释器中测试它。

(4) 留在 bitwise 文件夹中，启动 Python 并输入下面的代码：

```

>>> import bitmask
>>> bm1 = bitmask.BitMask()
>>> bm1
0
>>> bin(bm1.NOT() & 0xf)
'0b1111'
>>> bm2 = bitmask.BitMask(0b10101100)
>>> bin(bm2 & 0xFF)
'0b10101100'
>>> bin(bm2 & 0xF)
'0b1100'
>>> bm1.AND(bm2)
0
>>> bin(bm1.OR(bm2))
'0b10101100'
>>> bm1 = bm1.OR(0b110)
>>> bin(bm1)
'0b110'
>>> bin(bm2)
'0b10101100'
>>> bin(bm1.XOR(bm2))
'0b10101010'
>>> bm3 = bm1.shiftleft(3)
>>> bin(bm3)
'0b110000'
>>> bm1 == bm3.shiftright(3)
True
>>> bm4 = bitmask.BitMask(0b11110000)
>>> bm4.listbits()
[1, 1, 1, 1, 0, 0, 0]
>>> bm4.listbits(2,5)
[1, 1, 0]
>>> bm4.listbits(2, -2)
[1, 1, 0, 0]

```

(5) 退出解释器。

既然你已经证明新模块可以正常工作，那就可以继续把 bitwise 目录转换成一个 Python 包。

(6) 新建一个空的 `__init__.py` 文件。

(7) 为了测试包是否可以正常工作，需要将你的工作目录改成 bitwise 的上一级目录。

现在就做这件事。

现在需要测试你能否导入包和它的内容，并访问它的功能。

(8) 启动 Python 解释器并输入下面的测试代码：

```
>>> import bitwise.bits as bits
>>> from bitwise import bitmask
>>> bits
<module 'bitwise.bits' from 'bitwise/bits.py'>
>>> bitmask
<module 'bitwise.bitmask' from 'bitwise/bitmask.py'>
>>> bin(bits.AND(0b1010,0b1100))
'0b1000'
>>> bin(bits.OR(0b1010,0b1100))
'0b1110'
>>> bin(bits.NOT(0b1010))
'-0b1011'
>>> bin(bits.NOT(0b1010) & 0xFF)
'0b11110101'
>>> bin(bits.NOT(0b1010) & 0xF)
'0b101'
>>> bm = bitmask.BitMask(0b1100)
>>> bin(bm)
'0b1100'
>>> bin(bm.AND(0b1110))
'0b1100'
>>> bin(bm.OR(0b1110))
'0b1110'
>>> bm.listbits()
[1, 1, 0]
```

示例说明

创建了一个基于内置整数类型 `int` 的类。由于仅仅为类提供新方法而不存储任何额外的数据特性，因此不需要提供一个 `__new__()` 构造函数或 `__init__()` 初始化函数。这些方法与 `bits.py` 中写的函数非常类似，但是这里你创建了一个 `BitMask` 实例作为返回类型。`listbits()` 方法也展示了另一种获得位列表的方法。它首先使用 `bin()` 把值转化为字符串表达式。然后进行列表推导而创建一个列表，它利用了 `int()` 进行字符转整型的转换。同时，`listbits()` 方法也被扩展并接受一对 `start` 和 `end` 参数。这对参数的默认值就覆盖了这个二进制数的全部长度。但是它不能被用来获得位的子集。这里面有一小部分的工作是根据索引的正负来调整 `end` 的值。负的下标值不需要加两个字符，因为它们会自动地从右边索引。这样就使用了一个 Python 条件赋值语句来保证 `end` 值是正确的。

既然已经创建了类，那么就可以把它从本地目录中导入，并把它作为一个标准模块进行测试。随后，你重复了一组与对 `bits.py` 进行的测试相类似的测试。值得注意的一点是，可以混合和匹配传统按位操作符和新的函数版本。正如你在 `shiftright()` 示例中看到的那样，也可以就像任何其他整型那样去比较 `BitMask` 对象。最后，你证实新的 `listbits()` 算法可以

正常工作，并且新增的额外参数也可以在正值和负值的情况下都如预期的那样工作。

在这个阶段，你已经在文件夹中创建了两个标准模块。然后，创建了一个空白的 `__init__.py` 文件，把这个文件夹转变成为一个 Python 包。为测试它能否正常工作，回到上一级目录。这样，这个包对于 Python 解释器就是可见的了。然后，你确认可以导入包、模块，并访问一些功能。恭喜，现在你已经拥有了一个包含两个模块的包。

了解如何创建和使用标准的以及那些你自己创建的模块和包是一个了不起的开端。然而，网上还有更多可用的模块和包在等待下载。下一节将要解释你该如何做。

1.8 使用第三方包

在 Python 中，很多第三方的包都是可用的。很多这些包的二进制发布包连同安装程序对于大多数常用的操作系统都是可用的。如果一个二进制安装包可用的话，或者在包网站上或者对于 Linux 用户来说在你的包管理工具中，你应该使用它。因为这是让事情启动并运行的最简单的方式。如果一个二进制包不可用，则需要下载并安装基本包。

可在 Python 包索引(Python Package Index, PyPI)中找到很多这些第三方包。PyPI 的网站是 <https://pypi.python.org/pypi>。它们以一种特殊的格式被分发，这样它本身也需要安装一个第三方包。这种先有蛋还是先有鸡的情况经常会困扰初学者，所以本节描述如何搭建可以访问这些第三方包的环境。

PyPI 包附带了一种叫做蛋(egg)的格式。一个 Python 蛋既可以传送一个标准 Python 包，也可以传送一个用 C 语言或 Python 和 C 代码混合编写的二进制包。

Python 包装的未来：

蛋格式有很多问题，并且它自己也正在被称为轮子(wheel)的东西替代。这是更广泛的策略的全部。它想要让多种发布 Python 和应用的方法合理化。Python Package Authority 正在主导这个项目。最终，所有创建和安装 Python 包所需要的工具都应该在一个标准的 Python 安装中。这个路线图开始在 Python 3.4 中起作用了，它把 pip 包含在标准的发布中。

如果想要创建自己的可发布包，应该去 Python Package Authority 的网站(<https://python-packaging-user-guide.readthedocs.org/en/latest/>)上查看最新指导。

安装一个蛋需要一个称为 pip 的工具。幸运的是，安装 pip 并不需要 pip。在 Python 3.4 版本中，pip 被包含在标准库中，这在一定程度上简化了流程。如果 pip 没有被包含在你的 Python 版本中，可以自己安装 pip。浏览网站 https://pip.pypa.io/en/latest/reference/pip_install.html 并按照说明去做。

通过网页上的链接将 `get-pip.py` 文件下载到你电脑中任意方便的文件夹中。把工作环境换到那个文件夹，确认你连接到了互联网，并且执行下面的命令：

```
python get-pip.py
```

这可能会要花一些时间并且从互联网下载一些东西。你将会看到一些类似这样的消息：

```
$ python3 get-pip.py
Downloading/unpacking pip
  Downloading pip-1.5.2-py2.py3-none-any.whl (1.2MB): 1.2MB downloaded
Installing collected packages: pip
Successfully installed pip
Cleaning up...
```

一旦 `pip` 被安装好，就可以使用它来安装 `PyPI` 包，使用下面这个命令：

```
pip install SomePackageName
```

这个命令会安装指定包的最新版本。也可以用这个命令轻松卸载一个包：

```
pip uninstall SomePackageName
```

还有很多其他的选项可以使用，它们在 `pip` 文档页有详细的描述：<https://pip.pypa.io/en/latest/reference/index.html>。

并非所有的包都使用 `pip`，同时也存在其他的安装选项和工具。包文档应该解释需要做什么。

1.9 本章小结

在本章中回顾了 `Python` 的核心语言特性，了解了解释器环境、核心数据类型，以及语言控制结构和语法。也知道了如何通过写函数、类、模块和包来扩展 `Python`。

核心数据类型包括布尔(`bool`)、整数(`int`)、浮点数(`float`)以及特殊的 `None` 类型。`Python` 也支持多个容器类型，包括字符串、字节、列表、元组、字典和集。

控制结构涵盖了所有的结构化编程概念，包括序列、选择和迭代。选择是通过使用 `if/elif/else` 和一个条件表达式来完成的。有两种循环结构体支持迭代：`for` 和 `while`。`for` 会迭代一个集合或可迭代对象。`while` 则更加普通，还可能产生无限循环。`Python` 也通过 `try/except/finally` 结构体支持异常管理。

除了大量的内置函数和标准模块库，`Python` 也允许你通过使用 `def` 或 `lambda` 关键字写自己的函数来扩展它的功能。也可以创建自己的数据类型并创建这些类的实例来扩展标准数据类型。类的定义使用 `class` 关键字。函数和类可以保存在单独的文件中。这样就构成了 `Python` 模块。它们可以被导入到其他代码中，因此方便了跨程序复用。模块还可以被整理成包。包其实就是包含 `__init__.py` 文件的文件夹。

练习题

1. 如何在不同的 `Python` 数据类型间进行转换？在转换数据类型时，会产生什么数据质量问题？

- 2. 哪些 Python 容器类型可以作为字典中的键使用？哪些 Python 数据类型可以作为字典中的值使用？
- 3. 使用一个 if/elif 链编写一个示例程序，需要包含至少 4 个不同的选择表达式。
- 4. 编写一个 Python for 循环，重复一个消息 7 次。
- 5. 如何使用一个 Python while 循环创建一个无限循环？这可能会产生什么问题？如何解决这个问题？
- 6. 编写一个函数，计算一个给定底和高的三角形面积。
- 7. 编写一个类，并实现一个从 0 到 9 的循环计数器。也就是说，这个计数器从 0 开始，一直递增到 9，再被重置为 0，这样无限地重复这个循环。该类应该有 increment()和 reset()方法。reset()方法会返回当前的计数，然后将计数设回为 0。

练习题的答案在附录 A 中。

本章所学知识

主 题	关 键 概 念
Python 基础结构	在无参数的情况下可以调用 Python 解释器，得到一个交互式 shell。如果把一个文件名作为参数，解释器会执行它并退出
简单数据类型	Python 支持整型、浮点型、布尔型和 None 数据类型。类型名可以被用作类型转换函数。Python 的类型是对象，并且支持很多操作
容器数据类型	Python 支持 Unicode 和字节字符串，以及列表、元组、字典和集。字符串和元组是不可变的(不可以被修改)，而字典和集需要不可变的类型作为键。大多数集合都是可迭代的，并且可以用在 for 循环中
基本控制结构	Python 支持序列、选择和重复。序列只是简单的代码行；不需要块标识符或语句结束符。可以通过 if/elif/else 结构来进行选择。Python 提供了两种循环：for 和 while。 代码块是通过前一行结尾的冒号标明的。代码块在行下面需要被缩进。重新回到原来的缩进级别意味着代码块的结束
错误处理	Python 通过 try/except/else/finally 结构来支持异常处理。 用户可以定义自己的异常或为内置的错误添加参数
输入/输出	使用 input()函数，可以将用户输入作为字符串读取。 通过 print()函数，用户输出可以被显示出来。 文本文件可以被打开、读取和写入。使用 tell()和 seek()可以实现文件导航
定义函数	使用 def 或 lambda 关键字可以定义新函数。函数可以通过参数来接收输入，通过 return 关键字提供结果
定义类	类是使用关键字 class 定义的。类支持单继承和多重继承、多态、操作符重载和方法重写。类特性可以被当成属性和/或插槽 特性可以通过点符号来访问。类也是对象

(续表)

模块和包	模块只是包含 Python 代码的、任何被列在 <code>sys.path</code> 中文件夹内的文件。包就是包含一些模块和一个名为 <code>__init__.py</code> 文件(可能为空)的文件夹。包也是模块。模块中的名称是通过点符号访问的
------	---

Python 脚本

本章主要内容:

- 通过操作系统访问和管理计算机资源
- 处理常用文件格式, 比如 CSV 和 XML
- 操作日期和时间
- 应用自动化和访问它们的 API
- 在标准库的功能之外, 使用第三方模块来扩展自动化

从 wroox.com 下载本章代码

可以在 wroox.com 网站找到本章涉及的代码。代码可以在 www.wroox.com/go/python-projects 的 Download Code 选项卡下找到。第 2 章代码位于 Chapter 2 downloaded, 名为 Chapter2.zip。每个文件都是根据本章提到的代码文件名命名的。

有时, 你可能发现自己在处理一些涉及很多重复操作的任务。为了减轻重复工作, 我们可以在单个应用里编写或求自动化那些操作。但是, 如果操作跨越多个应用, 就变得难有帮助了。例如, 如果你有一个大的多服务器 web 应用进行备份和存档, 则可能必须去处理一个或多个媒体工具产生的内容、集成开发环境的代码和可能的数据库文件。相比之下, 你更需要一个外部的编程工具来为每个应用或工具寻找并执行它们各自的工作。

Python 非常适合这种类似于管家或协调员的角色。

什么是脚本?

脚本是指, 1) 为自动化而编写的程序, 2) 为自动化而编写的程序, 3) 为自动化而编写的程序。

脚本是指, 1) 为自动化而编写的程序, 2) 为自动化而编写的程序, 3) 为自动化而编写的程序。

脚本是指, 1) 为自动化而编写的程序, 2) 为自动化而编写的程序, 3) 为自动化而编写的程序。

脚本是指, 1) 为自动化而编写的程序, 2) 为自动化而编写的程序, 3) 为自动化而编写的程序。

第 2 章

Python 脚本

本章主要内容:

- 通过操作系统访问和管理计算机资源
- 处理常见文件格式, 比如 CSV 和 XML
- 操作日期和时间
- 应用自动化和访问它们的 API
- 在标准库的功能之外, 使用第三方模块来扩展自动化

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/python-projects 的 Download Code 选项卡下找到。第 2 章代码位于 Chapter 2 download, 名为 Chapter2.zip, 每个文件都是根据本章提到的代码文件名命名的。

有时, 你可能发现自己在处理一些涉及很多重复操作的任务。为了避免重复工作, 我们可以在单个应用里编写宏来自动化那些操作。但是, 如果操作跨越多个应用, 宏就很难有效了。例如, 如果对一个大的多媒体 Web 应用进行备份和存档, 则可能必须去处理一个或多个媒体工具产生的内容、集成开发环境的代码和可能的数据库文件。相比于宏, 你更需要一个外部的编程工具驱动每个应用或工具来执行它们各自的工作。

Python 非常适合这种类似于管弦乐编曲的角色。

什么是脚本?

对于不同的人来说, 脚本意味着不同的事物。所以在这章看到它之前, 我们必须要先明确它的意思。它协调其他程序或应用的行为来执行一个任务, 比如批量文件打印、或一个自动化工作流程, 比如增加一个新用户。你可能在使用操作系统工具或类似办公生产力套装的大型通用包。想象一下, 戏剧中的剧本告诉演员台词、如何站位以及何时进出场

的方式。这就是 Python 脚本所做的。它协调其他程序的行为。

在本章，你会学到如何使用 Python 模块检查用户设置以及目录和文件的访问级别，为一个操作设置正确的环境，以及在脚本中启动和控制外部程序。你也会学到 Python 模块如何帮助你访问常用文件格式中的数据、如何处理日期和时间，以及如何使用非常强大的 ctypes 模块和专门用于 Windows 的 pywin32 包直接访问外部应用的低级别编程接口。

特定系统问题

从讨论的本质上来说，本章大部分的内容都是特定操作系统的。一些模块试图可移植到多个操作系统，然而其他模块在一些系统上只支持部分操作，还有一些对特定平台维护特定的模块。如果发现不好用的示例，确保它不是专为另一个不同的操作系统工作的。可能的话，我们会试着指出哪些模块是专为特定系统的。

同样，在“试一试”部分，输出通常也是特定于用户和操作系统的。所以不要期待会获得一模一样的结果。例如，在检查用户详情时，你应该会看到你自己的用户名，而不是示例中的用户名。

2.1 访问操作系统

大部分典型程序员使用操作系统执行的一些任务，比如收集用户信息或浏览文件系统，都可以使用 Python 的标准库模块以通用的方式完成(回想一下，模块是在多个程序间分享的可复用代码片段)。主要的模块都是以这种方式编写的：单个操作系统行为的特殊性都被隐藏在更高层的对象和操作之后。在本节中，你将研究模块 os/path、pwd、glob、shutil 和 subprocess。这些资料将专注于如何在一些常见的场景中使用这些模块。它并没有试图涵盖所有可能的情况或选项。

顾名思义，os 模块为许多操作系统特性提供访问。实际上，它拥有 os.path 子模块。这个子模块负责管理文件路径、名称和类型。在逐步完成本章的各种主题时，你会遇到 os 模块的其他子模块。这些不计其数的模块被整体地称为 OS(大写)模块，而实际的 os 模块被称为 os(小写)。如果熟悉 UNIX 系统编程，或者甚至使用 UNIX shell，比如 Bash，你会熟悉许多这样的操作。

操作系统主要用来管理计算机硬件的访问，包括 CPU、内存、存储和网络。它管理这些资源的访问以及进程的创建、调度和销毁。OS 模块函数可以了解和管理这些操作系统行为。接下来，你会看到如下常见的任务：

- 收集用户和系统信息
- 管理进程
- 确定文件信息
- 操纵文件
- 浏览文件夹



注意：Python 版本 3 中最大的改变是它持续地关注 Unicode 的国际化、识别和使用。os 模块也不例外。它的函数接受 Unicode 字符串。然而，底层的 OS 函数和文件名并不需要处理 Unicode。在那些情况下，Python 会使用 `sys.getfilesystemencoding()` 提供的系统编码把它转换为字节。这并不保证在所有情况下都好用，偶尔可能会抛出一个 `UnicodeError` 异常。在那些情况下，或者必须在调用函数之前把字符串转换成一个更加操作系统友好的格式，或者寻找其他方法解决问题。

2.1.1 获得关于用户和他们的电脑的信息

在探索 OS 模块时，首先要找出它们能够告诉你哪些用户信息。具体地说，可以找出用户 ID、登录名和一些默认设置。

就像 Python 中的大多数新事物一样，熟悉它们的最佳方式就是通过交互式提示符。所以启动 Python 解释器来试一试吧。

试一试：确定用户

在这个“试一试”中，你会找出当前用户的一些信息。完成以下步骤：

- (1) 启动 Python 解释器。
- (2) 在解释器中输入下面的代码：

```
>>> import os
>>> os.getlogin()
'agauld'
>>> os.getuid()                # Not Windows
1001
>>> import pwd                 # Not Windows
>>> pwd.getpwuid(os.getuid()) # Not Windows
pwd.struct_passwd(pw_name='agauld', pw_passwd='unused', pw_uid=1001,
pw_gid=513, pw_gecos='Alan Gauld,U-DOCUMENTATION\\agauld,
S-1-5-21-2472883112-933775427-2136723719-1001',
pw_dir='/home/agauld', pw_shell='/bin/bash')
>>> for id in pwd.getpwall():
...     print(id[0])
...
SYSTEM
LocalService
NetworkService
Administrators
TrustedInstaller
Administrator
```

```
agauld
Guest
HomeGroupUser$
???????
>>>
```

示例说明

在第一行导入 `os` 模块之后，你获得了登录名称字符串。通常这对于创建个性化的提示语和屏幕信息来说是最有用的。遗憾的是，对于 Windows 用户来说，到此为止了。其他部分代码只适合基于 UNIX 的系统。然而，也并非一无所有，因为也可以从环境变量中获得一些这样的信息。你会在 2.1.2 节了解到这些。

如果有一个基于 UNIX 的系统，那么使用 `os.getuid()` 可以获得操作系统产生的用户 ID，即一个数字值。之后可以在各种其他函数中使用它。下面代码导入了密码模块 `pwd` 并使用其中的函数将操作系统用户 ID 翻译成一组更加完整的信息，包括真实名称、默认 shell 以及主目录。很显然这样信息量更大。但是它需要使用首先从 `os.getuid()` 获得的 UID。另一个可用的函数 `os.getpwnam()` 接受登录名作为参数并返回同样的信息。最后，你使用 `pwd.getpwall()` 和一个 `for` 循环提取系统的所有用户名。

接下来，你会看到用户在他们创建的文件上有哪些权限。这是非常重要的，因为它影响着任何代码生成的文件。你可能需要暂时改变权限。例如，如果需要创建一个之后在程序中执行的文件，它需要有执行权限。在 UNIX 中，这些设置都被保存在一个名为 `umask` 或用户掩码的东西中。它是一个位掩码，就像你在第 1 章最后使用的那些。每一位代表着一个用户访问数据点，如下所述。

Python 允许你使用 `os.umask()` 函数查看 `umask` 值，甚至在 Windows 上也可以。但是，`os.umask()` 函数有一个小怪癖。它期待你向函数传入一个新值。然后它会设置这个值并返回旧值。但如果只是想找出当前的值，就不能这样做。相反，需要将 `umask` 设置为一个临时值，读取旧值，然后再将它设置为原始值。掩码的格式非常紧凑，包含三组三位的值，每一组分别对应着 Owner、Group 和 World 权限。



注意：3 位可以被简洁地表示为八进制数。出于这个原因，你经常会发现 UNIX 文档使用八进制数表示这些值。通过在八进制数前面加上 `0o` 来表示它们，Python 也可以使用八进制值，所以 `0o777` 表示所有位都为 1 的 9 位。

在组内，3 位分别对应着每一种访问——读、写或执行。这些是用显式的二进制符号最方便地表示出来。表 2-1 展示了每一个 3 位二进制值是如何映射到权限的。

表 2-1 Umask 二进制映射

Umask 二进制值	读，写，执行的值
000	读=True，写=True，执行=True
001	读=True，写=True，执行=False
010	读=True，写=False，执行=True
011	读=True，写=False，执行=False
100	读=False，写=True，执行=True
101	读=False，写=True，执行=False
110	读=False，写=False，执行=True
111	读=False，写=False，执行=False

既然你已经理解你想要做的事情，是时候试一试了。

试一试：读取和修改 umask 值

在这个“试一试”中，你将要读取、修改和恢复当前用户的 umask 值。完成以下步骤：

- (1) 启动 Python 解释器。
- (2) 在解释器中输入下面的代码：

```
>>> import os
>>> os.umask(0b11111111) # binary for all false - 111 x 3
18
>>> bin(18)
'0b10010'
>>> os.umask(18)
511
```

示例说明

首先，你使用了一个二进制值 11111111 来调用 os.umask()。这会设置所有权限为假。这样做是为了防止万一有什么事情发生错误。相比于让用户更容易受到安全漏洞攻击，还是为掩码增加限制比较好。

然后，Python 打印出了十进制值 18。通过调用 bin()函数，你看到 18 的二进制掩码值是 10010。如果用 0 填充把它变成一个完整的 9 位掩码并把它分割成 3 位的组，就会看到 000-010-010。查看表 2-1，你会发现这个值表示对 Owner 的完全访问权限，但是对 Group 和 World 用户只有读和执行权限。

最后，通过调用参数 18(umask 返回的原始值)的 os.umask()，你还还原了用户的初始设置，而之前设置的掩码值(11111111)被打印出来——十进制的 511。

有时，你想知道用户在运行什么样的操作系统，尤其是操作系统本身的细节。Python 有几种办法完成它，但是你首先会看到 os.name 属性。在写本文时，这个属性返回了下面值中的一个：posix、nt、mac、os2、ce 或 java。



注意：尽管严格来说并不是一个操作系统，但值 `java` 也被包含其中。对于 Java 版本 Python(默认是 C 版本的)，知道你正在一个 Java 虚拟机上运行是非常有用的。

另一个查找用户正在使用的系统的地方在 `sys` 模块中的 `sys.platform` 特性中。这个特性返回的信息通常与 `os.name` 返回的信息稍有不同。比如 Windows 被报告为 `win32` 而不是 `nt` 或 `ce`。在 UNIX 中，另一个 `os` 中名为 `os.uname()` 的函数提供了稍微详细些的细节。如果有多个不同操作系统可用，比较这些不同的技术返回的结果会非常有意思。推荐你只是使用 `os.name` 选项，因为它是普遍可用的，并且会返回一组定义明确的结果。

另外一个往往很有用的信息片段是用行和列表示的用户终端的大小。可以使用这个信息修改脚本中展示的消息。`shutil` 模块为此提供了一个名为 `shutil.get_terminal_size()` 的函数，如下所示：

```
>>> import shutil
>>> cols, lines = shutil.get_terminal_size()
>>> cols
80
>>> lines
49
```



注意：在老版本 Python 中，必须使用底层的 `os.get_terminal_size()`。但是在版本 3 中，推荐使用 `shutil` 版本。

如果不能确定终端大小，则默认返回值是 `80 x 24`。通过一个可选参数，可以指定为不同的默认值。但是 `80 x 24` 通常是一个明智的选择，因为它是终端模拟器的传统大小。

2.1.2 获得当前进程信息

对于一个程序来说，知道一些有关它当前状态和运行环境的信息是非常有用的。例如，你可能想要知道进程身份或进程是否有一个首选文件夹写入它的数据文件或读取配置数据。OS 模块为确定这些值提供了函数。

一个进程信息源就是通过环境变量定义的进程环境。`os` 模块提供了一个名为 `os.environ` 的字典，它为当前进程保存了所有环境变量。



注意：给定系统的变量数量取决于各种各样的本地情况，包括应用的数量和属性。因为许多应用在安装过程中会创建它们自己的环境变量值。

环境变量的缺点是它们非常不稳定。用户可以创建和移除它们。应用也是如此，所以依赖一个环境变量是非常危险的。你应该永远保持一个可以转而使用的默认值。幸运的是，一些值是相当可靠的而且通常是存在的。这其中有三个值对于 Windows 用户尤为有用，因为之前讨论的 `pwd.getpuid()` 和 `os.uname()` 函数在 Windows 下是不可用的。它们是 `HOME`、`OS` 和 `PROCESSOR_ARCHITECTURE`。

如果试图去访问一个没有定义的变量，你会得到一个常见的 Python 字典 `KeyError`。在大多数而不是全部操作系统上，一个程序可以设置或修改环境变量。如果操作系统支持这个特性，Python 会把任何 `os.environ` 字典的变化都反映到操作系统环境中。除了使用环境变量作为用户信息来源，也常使用它们定义程序的特定用户的配置细节，例如数据库的位置。现在，不是很赞成这种做法，而为这些细节信息使用配置文件被认为是更好的选择。但如果在使用一个较老的应用，你可能需要从环境中获得这些东西。

试一试：调查进程环境

在这个“试一试”中，你会调查你计算机上的进程环境。完成下面的步骤：

- (1) 启动 Python 解释器。
- (2) 在解释器中输入下面的代码。

```
>>> import os
>>> os.getpid()
16432
>>> os.getppid()
3165
>>> os.getcwd()
/home/agauld
>>> len(os.environ)
48
>>> os.environ['HOME']
'/home/agauld'
>>> os.environ['testing123']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/UserDict.py", line 23, in __getitem__
    raise KeyError(key)
KeyError: 'testing123'
>>> os.environ['testing123'] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/os.py", line 471, in __setitem__
```



```

    putenv(key, item)
TypeError: str expected, not int
>>> os.environ['testing123'] = '42'
>>> os.environ['testing123']
'42'
>>> del(os.environ['testing123'])
>>> os.environ['testing123']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/UserDict.py", line 23, in __getitem__
    raise KeyError(key)
KeyError: 'testing123'

```

示例说明

在第一行导入 `os` 之后，你做的第一件事就是确定当前进程 ID。由于你在 Python 解释器中，因此得到的是解释器进程本身。下一行读入了父进程的 ID，在本例中，它是操作系统 shell。当与其他操作系统工具交互时，可以使用这些 ID，详见下一节描述。

然后你使用 `os.getcwd()` 函数(它代表着获取当前工作目录)来确定哪个目录是当前的默认目录。这通常是指解释器的启动目录，但是你将看到，在程序中可以更改目录。`os.getcwd()` 是一个在任意时间精确地检查代码工作目录的非常有用的方式。

然后，通过展示 `os.environ` 字典的长度，你看到了环境的大小。然后你取出了 HOME 环境变量的值，它展示了用户的主目录。之后，你试图去访问名为 `testing123` 的变量，但是由于它并不存在，你得到了 `KeyError`。你试图通过赋值数字 42 来创建一个 `testing123` 变量，但是产生了一个 `TypeError`，因为环境变量必须是字符串。然后你把值 '42' 复制给了 `testing123` 变量，并且成功地为这个进程创建了一个新的环境变量(切记，它是这个进程和任何子进程的局部环境。外部进程是看不到它的)。然后，再次读取它的值，这一次你得到了值，没有任何错误信息。这证明你已经成功了。最后，删除了变量，并且通过再次试图读取它并获得期待的错误来证明它已经不存在了。

2.1.3 管理其他程序

能够在一个脚本中运行其他程序通常是很有用的，而 `subprocess` 模块是首选工具。`subprocess` 模块包含一个名为 `Popen` 的类，它提供了一个非常强大而灵活的外部程序接口。当需要更简单的方法时，可以使用这个模块提供的几个便利函数。文档描述了如何使用所有这些特性。在本节中，只会使用最简单的函数 `subprocess.call()` 和 `Popen` 类。



注意：在历史上，Python 已经有多种方式运行子进程。这已经引起相当大的混乱和不断增加的选项。大多数机制仍然可用，以免破坏旧的代码。但是强烈建议你任何新代码中使用这里讨论的 `subprocess` 模块。

`subprocess` 模块最基本的用途是调用外部操作系统命令，并只是让它以自己的方式运行。输出通常会被展示在屏幕上或存储在某处的数据文件中。在程序结束后，可以让用户基于展示的内容做一些选择，或者可以在代码中直接访问数据文件。尤其是在基于 UNIX 的系统上，可以通过提供适当的命令行选项或使用操作系统文件重定向来强制许多操作系统工具输出到一个数据文件。这个技术用于驾驭操作系统工具是非常强大的。Python 可以使用它做进一步处理。

这种调用程序的基本机制被包装在 `subprocess.call()` 函数中。这个函数接受一个字符串列表作为它的第一个参数，以及几个可选的关键字参数。这些可选参数可以被用来控制输入和输出位置，以及一些其他的事情。

下面在“试一试”中查看它的工作原理。

试一试：启动外部程序

在这个“试一试”中，你会从 Python 中调用各种各样的程序。完成下面的步骤：

(1) 创建一个名为 `root` 的测试目录，并且用少量的文本文件填充它(或者重新创建，或者从别的地方复制它们)。它们的内容并不重要，在这个阶段你只是对它们的所在感兴趣。为了获得与展示相同的结果，结构应该如下所示：

```
root
fileA.txt
fileB.txt
```

(2) 切换到 `root` 目录，并启动 Python 解释器。

(3) 在 `>>>` 提示符下输入下面的代码(一定要为你的操作系统使用正确的命令)：

```
>>> import subprocess as sub
>>> sub.call(['ls']) # Not Windows
fileA.txt fileB.txt
0
>>> sub.call(['ls'], stdout=open('ls.txt', 'w')) # Not windows
0
>>> sub.call(['cmd', '/c', 'dir', '/b']) # Windows only
fileA.txt
fileB.txt
0
>>> sub.call(['cmd', '/c', 'dir', '/b'], stdout=open('ls.txt', 'w'))
# Windows only
0
>>> sub.call(['more', 'ls.txt']) # Not Windows
fileA.txt
fileB.txt
ls.txt
0
>>> sub.call(['cmd', '/c', 'type', 'ls.txt']) # Windows only
fileA.txt
fileB.txt
```

```
ls.txt
0
>>> for line in open('ls.txt'): print(line)
...
fileA.txt
fileB.txt

ls.txt
```

示例说明

在第一行以别名 `sub` 导入 `subprocess` 之后(这只是为了后面节省一些输入量),你第一次调用了 `sub.call()`, 参数是 `['ls']` 或 `['cmd','/c',['dir'],'/b']`。这取决于你的操作系统(注意 `dir` 实际上是 `cmd.exe` 进程的一个子命令。Windows 帮助系统解释了 `/c` 和 `/b` 标志是干什么的)。输出被展示在 `stdout`(你的终端)上,但是 Python 无法访问文件名。`call()` 的唯一返回值就是操作系统的返回码,它告诉你程序是否成功地完成。但是它不能帮助你进行数据交互。然后你第二次使用了 `sub.call()`,但是这一次你把 `stdout` 重定向到一个新的文件: `ls.txt`。然后你使用操作系统工具 `more`(或 Windows 的 `type`)来展示文件。`ls.txt` 是一个普通的文本文件。这意味着可以使用 Python 命令以常规的方式打开并处理文件来访问数据。在这个情况下,你只是循环每行并打印它们,但是也可以只是把数据存储起来以作他用。值得注意的是,像这样将文件列表暴露在文本文件中存在潜在的安全问题,所以你应该在处理它之后尽快删除文件。



提示: 注意,这里很可能会混淆终端会话,尤其是在后台运行进程时。结果通常是键盘看起来停止响应或者产生奇怪的字符。如果这种事情发生了,恢复秩序最简单的方法就是使用操作系统工具消灭任何错误的进程。这是 Python 运行子进程的危险之一。`subprocess` 模块会尽量使你免受这些情况的困扰,但是你有时也可能因为疏忽而把事情搞砸。

运行外部程序时可能发生的一个问题是操作系统找不到命令。一般来说,当它发生时,会得到一个错误消息。所以它真实存在,需要显式地提供程序文件的完整路径。

最后,我们研究一下如何停止运行中的进程。对于交互式程序的用户来说,关闭外部程序的最简单方式就是正常的方式或者发出一个中断信号——使用 `Ctrl+C` 或 `Ctrl+Z` 或者用户操作系统上的规范。但是对于非交互式的程序,你可能需要从操作系统内部进行干预——通常是检查运行进程的列表并显式地终止错误的进程。



注意：在 Windows 上，另一个有用的工具是 `os.startfile()`。你传一个文件名而不是传命令给函数，然后 Windows 使用它的文件关联数据库来启动适当的命令。例如，如果传过去一个文本文件，它可能会启动 Notepad 编辑器。第二个可选的参数称为 `operation`，它规定了被调用的程序应该如何处理文件。最常用的选项就是默认的 `open` 或者 `print`。被指定的操作必须被 Windows 认为是这个文件类型的操作。可以在 Windows Explorer 中右击文件来查看这些。然而，只有那个列表的一个子集是与外部程序关联的。一个非法操作会抛出 `OSError` 异常。如果有疑问，则使用 Python 解释器试试这些选项并找出哪些可以正常工作。

你已经看到使用 `subprocess.call()` 启动一个外部进程是多么的容易。现在你要学习 `subprocess` 模块如何帮助你更多地控制进程，尤其可以帮助程序与正在运行的线程进行交互，特别是如何直接在你的脚本里读取进程输出。

2.1.4 更加高效地管理子进程

可以使用 `Popen` 类创建一个进程或命令的实例。遗憾的是，文档看起来相当令人气馁，因为 `Popen` 构造函数有相当多的参数。好消息是几乎所有的那些参数都有有用的默认值，而且在最简单的情况下是可以被忽略的。因此，如果只是为了在脚本内执行一个操作系统命令，只需要做如下操作(Windows 用户应该替换成之前示例中的 `dir` 命令)：

```
>>> import subprocess as sub
>>> sub.Popen(['ls', '*.'], shell=True)
<subprocess.Popen object at 0x7fd3edec>
>>> book tmp
```

注意 `shell=True` 语句。为了能够让命令被操作系统命令处理器或 `shell` 解释，这很有必要。这样做是为了保证通配符号(`.*`)以及任何字符串引用和类似的都会按照你期待的方式被解释。如果不使用 `shell` 参数，就会发生这种情况：

```
>>> sub.Popen(['ls', '*.'])
<subprocess.Popen object at 0x7fcd328c>
>>> ls: cannot access *.*: No such file or directory
```

由于没有指定 `shell=True`，因此操作系统试图去查找一个名称为`*.*`但不存在的文件。使用 `shell=True` 的问题是它会产生一个注入式攻击的潜在安全性问题。所以如果命令是动态创建的，比如根据文件中的或从用户输入的字符串，那么永远不要使用它。



注意：注入式攻击是这样一种情况：一个恶意的(或非常粗心的)用户打字输入一个字符串，这个字符串会被你的程序读取并解释。但是它包含了可能有害的命令而不是无害的数据。这可能会导致删除文件或更加恶劣的后果。shlex 模块包含了一个 quote() 函数，它可以减轻风险，但是在运行动态生成的字符串时，还是要十分小心。

为了访问运行命令的输出，可以在调用时添加两个额外的参数，如下：

```
>>> lsout = sub.Popen(['ls', '*.'], shell=True, stdout=sub.PIPE).stdout
>>> for line in lsout:
...     print (line)
```

这里，你指定 stdout 应该是一个 sub.PIPE，然后把 Popen 实例的 stdout 特性赋值为 lsout(一个管道(pipe)只是到另一个进程的数据连接。在这种情况下，它连接了你的程序和你正在执行的命令)。完成之后，可以把 lsout 变量就像普通 Python 文件那样对待，并且从它读取数据等。

通过指定 stdin 为你之后写入数据的通道，也可以用基本相同的方式向进程发送数据。可以向各种流赋值很多有效值，包括已经打开的文件、文件描述符或其他流(比如，这样可以让 stderr 出现在 stdout 上)。请注意，可以把第二个程序的输入设置为第一个程序的输出。这样可以把外部命令连接在一起。它与在命令行使用操作系统管道符号(|)的效果类似。

试一试：使用 subprocess.Popen 来访问 stdin/stdout

想要观察如何使用 subprocess.Popen 来与进程进行交互，请完成下面的步骤：

- (1) 在 root 文件夹中启动 Python 解释器。
- (2) 输入以下代码：

```
>>> import subprocess as sub
>>> sub.Popen(['ls']) # Windows use: ("cmd /c dir /b")
<subprocess.Popen object at 0x7fd3eccc>
fileA.txt fileB.txt ls.txt
>>> # Windows use: ("cmd /c dir /b", stdout=sub.PIPE)
>>> ls = sub.Popen(['ls'], stdout=sub.PIPE)
>>> for f in ls.stdout: print(f)
...
b'fileA.txt\n'
b'fileB.txt\n'
b'ls.txt\n'
>>> ex = sub.Popen(['ex', 'test.txt'], stdin=sub.PIPE) # Not Windows
>>> ex.stdin.write(b'i\nthis is a test\n.\n') # Not Windows
19
>>> ex.stdin.write(b'wq\n') # Not Windows
3
```

```
>>>
1+ Stopped python3
>>> sub.Popen(['NonExistentFile'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.2/subprocess.py", line 745, in __init__
    restore_signals, start_new_session)
  File "/usr/lib/python3.2/subprocess.py", line 1361, in _execute_child
    raise child_exception_type(errno_num, err_msg)
OSError: [Errno 2] No such file or directory: 'NonExistentFile'
```

示例说明

一开始，你使用别名 `sub` 导入了 `subprocess` 模块。前两个命令只是复制了你对 `subprocess.call()` 所做的。在这里你首先创建了一个文件并在 `stdout` 上列出，但是你同样不能使用那个数据。第二种情况更为有意思，因为你把 `stdout` 重定向到 `sub.PIPE`。它允许你通过 `stdout` 特性来读取它(注意 `stdout` 参数和 `stdout` 特性之间的区别。对于 `stdout` 参数，你在调用 `Popen` 构造函数时把它赋值为 `sub.PIPE`。而对于 `stdout` 特性，你用它从 `Popen` 实例中读取数据，并且通过点符号访问它)。为了使用这个，你也把 `Popen` 调用的结果赋值给一个名为 `ls` 的变量。`Popen` 实际上是一个类，而调用的结果是一个新的 `Popen` 对象实例。最后的结果与 `subprocess.call()` 示例非常相似。你把输出写入名为 `ls.txt` 的文件中，然后读取文件。但是在这里，你最终没有创建任何文件。相反，你直接从进程中读取它。这意味着你使用少量临时文件并不会打乱文件系统，临时文件需要在后来清理掉。

下面的示例使用了 UNIX 行编辑器 `ex`。但是这一次，将 `stdin` 重定向到 `sub.PIPE`，然后向编辑器填充一些命令来创建一个短文本文件。你也通过在 `Popen` 的第一个参数中提供了第二个字符串，将一个文件名作为一个参数。注意，`stdin` 的输入必须是一个字节字符串(`b'xxxx'`)而不是常规的文本字符串。

倒数二个示例展示了当试图去打开一个并不存在(或没有提供正确的路径信息)的文件时会发生什么。你得到了一个 `OSError` 异常，当然可以使用 Python 的 `try/except` 结构来捕捉该异常。

在“试一试”中，你直接访问了 `stdin` 和 `stdout`。然而，这有时会产生问题，尤其是当同时运行进程或在线程中导致管道填充并阻塞进程。为了避免这些问题，这里推荐你使用 `Popen.communicate()` 方法并且索引适当的流。这虽然用起来稍微有一些麻烦，但可以避免刚刚提到的问题。`Popen.communicate()` 接受一个输入字符串(相当于 `stdin`)作为参数，并且返回一个元组。元组的第一个元素就是 `stdout` 的内容，第二个元素就是 `stderr` 的内容。所以，使用 `Popen.communicate()` 重复列出文件的示例如下：

```
>>> ls = sub.Popen(['ls'], stdout=sub.PIPE)
>>> lsout = ls.communicate()[0]
>>> print(lsout)
b'fileA.txt\nfileB.txt\nls.txt\n'
>>>
```


在这部分结尾值得指出的是，你已经在示例中使用了非常基本的命令，例如 `ls`。许多这些命令也都可以在 Python 本身内部执行(你将很快看到)。像 `subprocess.call()` 和 `Popen` 这些机制的真实价值其实是运行更加复杂的程序，比如文件转换实用工具和图片批处理工具。在 Python 中实现与这些工具相同的功能会是一个非常大的工程。所以调用外部程序是更加明智的选择。你利用 Python 的强项，即组织和验证输入和输出，而把重活留给更加专业的应用。

2.1.5 获取文件(和设备)的信息

`os` 模块的工作方式更像 UNIX。同样，它对待设备和文件也是一样。所以找出设备的相关信息，比如当前终端会话，看起来和找出文件信息一样。在本节中，你会看到如何确定文件的状态和权限，以及如何从程序内部改变它们的一些属性。代码如下：

```
>>> import os
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'ls.txt', 'test.txt']
>>> os.stat('fileA.txt')
posix.stat_result(st_mode=33204, st_ino=1125899907117103,
st_dev=1491519654, st_nlink=1, st_uid=1001, st_gid=513,
st_size=257, st_atime=1388676837, st_mtime=1388677418,
st_ctime=1388677418)
```

在这里，使用 `os.listdir()` 检查了当前目录('.')的文件列表(既然你已经看到了 `os.listdir()`，你可能意识到在 `subprocess` 中使用 `ls` 或 `dir` 有些做作，因为 `os.listdir()` 直接在 Python 中做了相同的事，并且更加高效)。然后你使用 `os.stat()` 函数获得其中一个文件的信息。这个函数返回了一个指定的元组对象，它包含了 10 个有用的项。这里面最有帮助的可能就是 `st_uid`、`st_size` 和 `st_mtime`。这些值代表着文件 Owner 的用户 ID、大小和最后的修改日期/时间。时间是整数，必须用 `time` 模块解码，如下所示：

```
>>> import time
>>> time.localtime(1388677418)
time.struct_time(tm_year=2014, tm_mon=1, tm_mday=2, tm_hour=15,
tm_min=43, tm_sec=38, tm_wday=3, tm_yday=2, tm_isdst=0)
>>> time.strftime("%Y - %m - %d", time.localtime(1388677418))
'2014-01-02'
```

在此使用了 `time` 模块的 `localtime()` 函数把整数 `st_mtime` 的值转换成本地时间值的元组。然后使用了一个适当的格式化字符串把它转换成可读的日期字符串(将在本章的 2.2.1 节中更加清晰地了解 `time` 模块)。

从 `os.stat()` 返回的简单的 10-值元组通常是很方便的，但使用 `os.stat()` 函数可以获得比元组直接提供的信息更多的详情。这些额外数值中的一些依赖操作系统，比如在 RiscOS 系统上出现的 `st_obtype` 特性。你需要花更多时间来挖掘出这些内容。可以使用对象特性点记号来访问详情。

可能你从 `os.stat()` 获取到的最有意思的东西就是 `st_mode` 值，它会告诉你文件的访问权限信息。代码如下：

```
>>> import os
>>> stats = os.stat('fileA.txt')
>>> stats.st_mode
33204
```

但是帮助不大。它看起来只是一个随机数！秘密其实隐藏在这个数的每一位中，它其实是另一个位掩码。可以回想一下你在本章前面见到的 `umask` 位掩码。`st_mode` 在概念上和 `umask` 类似，但是位的意义是相反的。可以通过最后 9 位查看访问细节是如何被编码的，如下所示：

```
>>> bin(stats.st_mode)[-9:]
'111111101'
```

通过组合使用 `bin()` 函数和切片，你提取出了最后 9 位二进制数。通过把它们看成 3 个 3 位的组，可以分别看到 Owner、Group 以及 World 的读/写/执行的值。因此，在这个示例中，Owner 和 Group 都把所有的三位设为 1(真)，而只有 World 把读和执行位设为 1(真)，把写设为 0(假)。(注意，这些与 `umask` 位的意义是直接相反的，不要混淆它们！)

高阶层也有意义，而且 `stat` 模块包含了一组位掩码。可以用它们在逐位的基础上提取细节。对于大多数目的来说，前面的访问位已经足够了。而且 `os.path` 模块的辅助函数可以帮助你访问那个信息。你将在本章之后的 `os.path` 部分再次看到这个话题。

在 Python 中，还有很多其他方法来得到一个文件的访问权限，尤其是 `os` 模块的 `os.access()` 函数。这个函数接受一个文件名和标记变量(`os.F_OK`、`os.R_OK`、`os.W_OK` 或 `os.X_OK` 中的一个)作为参数并返回一个布尔结果。这个结果分别表示文件是否存在、是否可读、是否可写或者是否可执行。相比于底层的 `os.stat()` 和位掩码方法，这些函数更加易用，但是了解这些函数从哪里获得数据非常有用。

最后，`os` 文档指出：在打开文件之前检查访问权限可能存在一个潜在的问题。在两个操作之间有一段很短的时间，在这期间文件的访问权限或内容可能被修改。所以，就像在 Python 中常用的那样，最好使用 `try/except` 打开文件并且处理发生这个问题时产生的失败。如果需要，可以之后使用访问检查来确定失败的原因。推荐的模式如下：

```
try:
    myfile = open('myfile.txt')
except PermissionError:
    # test/modify the permissions here
else:
    # process the file here
finally:
    # close the file here
```

既然已经介绍了如何去探索单个文件的属性，那么现在就应该介绍遍历文件系统、读

取文件夹、复制、移动和删除文件等机制。

2.1.6 浏览和操纵文件系统

Python 为打开、读取和写入单独文件提供了内置函数。os 模块添加了一些函数，将文件作为实体来操作，比如重命名、删除和创建链接。然而对于使用文件来说，这仅仅是 os 模块一半的本事。os 模块 shutil 模块和其他实用工具模块一起工作时，你会看到另一半。



注意：Python 3.4 引入了一个名为 pathlib 的新模块，它想要为文件系统提供一个面向对象的视图。它可能替代了本部分讨论的大部分功能。然而，pathlib 被标记为临时的，这意味着接口在未来的版本中可能会有重大改变，或者甚至模块可能会从库里被删掉。出于不确定性的考虑，这里并没有使用 pathlib。

首先，读取和浏览文件系统。你已经看到了如何使用 os.listdir() 获得一个目录列表和 os.getcwd() 来获得当前工作目录的名称。可以使用 os.mkdir() 创建一个新的目录和 os.chdir() 来浏览一个不同的目录。

试一试：在 Python 中访问目录

在这个“试一试”中，使用 Python 创建并访问目录。完成下面的步骤：

- (1) 切换到你之前使用的根目录。
- (2) 启动 Python 解释器并输入以下 Python 命令：

```
>>> import os
>>> cwd = os.getcwd()
>>> print (cwd)
/home/agauld/book/root
>>> os.listdir(cwd)
['fileA.txt', 'fileB.txt', 'ls.txt']
>>> os.mkdir('subdir')
>>> os.listdir(cwd)
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir']
>>> os.chdir('subdir')
>>> os.getcwd()
'/home/agauld/book/root/subdir'
>>> os.chdir('..')
>>> os.getcwd()
'/home/agauld/book/root'
```

示例说明

在第一行导入 os 之后，你把当前目录存储在 cwd 中，然后打印它来确定你在正确的地方。然后你列出了它的内容。之后，你创建了一个名为 subdir 的目录并切换到那个文件夹。通过再次在新文件夹中调用 os.getcwd()，检查这个变动是否成功并且发现文件夹确实

已经改变了。最后，使用`..`快捷方式回到之前的目录，并且再一次使用 `os.getcwd()`来检查该操作是否成功。

这里用到的 `os.mkdir()`函数存在一个问题，那就是它只能在一个已经存在的目录中创建一个目录。如果试图在一个不存在的地方创建一个目录，它会失败。Python 提供了另一个名为 `os.makedirs()`的函数。注意拼写上的不同。它会创建所有路径中还不存在的中间文件夹。

可以使用下面的命令了解它的工作原理：

```
>>> os.mkdir('test2/newtestdir')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: 'test2/newtestdir'
>>> os.makedirs('test2/newtestdir')
>>> os.chdir('test2/newtestdir')
>>> print( os.getcwd() )
/home/agauld/book/root/test2/newtestdir
```

在这里，原始的 `os.mkdir()`调用产生了一个错误，因为文件夹 `test2` 并不存在。然而，调用 `os.makedirs()`却成功了，它同时创建了 `test2` 和 `newtestdir` 文件夹。可以通过切换到 `newtestdir` 文件夹来证明这一点。注意，如果目标文件夹已经存在，那么 `os.makedirs()`会抛出一个错误。可以使用两个额外的参数来进一步微调它的行为，但是默认值通常已经足够。

另一个模块 `shutil` 提供了一系列高级别的文件操作命令。这些功能包括复制单个文件、复制整个目录树、删除目录树和移动文件或整个目录树。删除单个或一组文件的功能有点反常。这实际上是以 `os.remove()`函数的形式存在于 `os` 模块中的(和专为空目录的 `os.rmdir()`函数，尽管 `shutil.rmtree()`更加强大，也通常是你想要的)。

另一个有用的模块是 `glob`。这个模块提供了文件名通配符处理。你可能会熟悉`?`和`*`通配符。它们用来在操作系统命令中指定一组文件。例如，`*.exe`指定了所有以`.exe`结尾的文件。`glob.glob()`会在代码中做同样的事情，它会根据一个给定的模式返回一组匹配的文件名列表。

试一试：使用通配符、复制、删除和移动文件

在这个“试一试”中，会使用 `os`、`glob` 和 `shutil` 中的函数来操纵整个文件。完成以下步骤：

- (1) 切换到之前创建的 `root` 目录。
- (2) 创建一个名为 `test.py` 的新文件。其中内容并不重要，你只是对它的名称感兴趣！
- (3) 启动 Python 解释器并且输入下面的代码：

```
>>> import os,glob,shutil as sh
>>> os.listdir('.') # everything in the folder
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py', 'test2']
>>> glob.glob('*') # everything in the folder
```

```
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py', 'test2']
>>> glob.glob('*.*) # files with an extension
['fileA.txt', 'fileB.txt', 'ls.txt', 'test.py']
>>> glob.glob('*.txt') # text files only
['fileA.txt', 'fileB.txt', 'ls.txt']
>>> glob.glob('file?.txt') # text files starting with 'file'
['fileA.txt', 'fileB.txt']
>>> glob.glob('*.??') # any file with a 2 letter extension
['test.py']
```

(4) 仔细查看这些不同的结果集，可以看到函数和参数的不同组合的效果。

(5) 输入下面的代码：

```
>>> sh.copy('fileA.txt', 'fileX.txt')
>>> sh.copy('fileB.txt', 'subdir/fileY.txt')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'fileX.txt', 'ls.txt', 'subdir', 'test.py',
'test2']
>>> os.listdir('subdir')
['fileY.txt']
>>> sh.copytree('subdir', 'test3')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'fileX.txt', 'ls.txt', 'subdir', 'test.py',
'test2', 'test3']
>>> os.listdir('test3')
['fileY.txt']
>>> sh.rmtree('test2')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'fileX.txt', 'ls.txt', 'subdir', 'test.py',
'test3']
```

(6) 回顾一下刚刚输入的命令的输出结果，在输入下面的代码前仔细考虑它们的影响：

```
>>> os.mkdir('test4')
>>> sh.move('subdir/fileY.txt', 'test4')
>>> os.listdir('test4')
['fileY.txt']
>>> os.listdir('subdir')
[]
>>> os.remove('test4/fileY.txt')
>>> os.listdir('test4')
[]
>>> os.remove('test4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 1] Operation not permitted: 'test4'
>>> sh.rmtree('test4')
>>> sh.rmtree('test3')
>>> os.remove('fileX.txt')
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py']
```

示例说明

在开始的一组命令中，对比了 `os.listdir()` 和 `glob.glob()` 函数的几个模式的效果。第一个模式 `(*)` 通过列出文件夹的全部内容重复了 `os.listdir()` 所做的工作。第二个模式 `(*.*)` 列出了所有带扩展名的文件(严格来说，`glob()` 不知道关于文件或文件夹的任何事情，它只是严格地使用名称，所以它实际上列出了所有包含句点的名称，而不管它到底是什么样的对象)。然后，使用了模式 `(*.*txt)` 来找出所有文本文件，又使用 `'file?.txt'` 找出任何名称以 `file` 开头后面跟着一个字符的任意 `txt` 文件。最后，使用了一个通配符的组合来找出任何以两个字符 `(*.*?)` 为扩展名结尾的文件。

第二组命令查看了 `shutil` 文件操作命令。

一开始，使用 `shutil.copy()` 来复制单个文件：将 `fileA.txt` 复制到相同文件夹中的名为 `fileX.txt` 的新文件，复制 `fileB.txt` 到子目录 `subdir` 的名为 `fileY.txt` 的新文件。然后使用两次 `os.listdir()` 在每个文件夹中查看结果。

然后，使用 `shutil.copytree()` 函数查看了目录级别的操作。这个函数把 `subdir` 目录和它的内容复制到了一个在进程中创建的新文件夹 `test3` 中。同样，两次使用了 `os.listdir()` 来确定结果。使用 `shutil.rmtree()` 来删除 `test2` 文件夹和它的内容。再一次，`os.listdir()` 验证了操作结果是正确的。

接下来，使用 `os.mkdir()` 创建了另一个新文件夹 `test4`。在这个新文件夹中，使用 `sh.move()` 移动了文件 `fileY.txt`。同样，在两个文件夹中都使用 `os.listdir()` 证明操作成功并且文件不在 `subdir` 中而在 `test4` 中。

最后，使用 `os.remove()` 查看了移除文件。第一个示例将文件从 `test4` 中移除并验证它已经被删除了。下一行代码试图移除 `test4` 本身，但是产生了一个错误，因为 `os.remove()` 只在文件上工作。如果需要移除某个目录，需要再次使用 `shutil.rmtree()` (或者可以使用 `os.rmdir()` 函数)。结束时，再次使用 `os.listdir()` 来确定文件夹已经不存在了。

如果查看 `shutil` 文档，会看到复制函数的有轻微不同行为的几个变种。在多数情况下，标准的 `shutil.copy()` 函数会如你所期。`shutil` 的其他特性包括创建 `zip` 或 `tar` 格式的归档文件或压缩文件。同样，可以使用可选参数来扩展这几个函数的功能。一个最有意思的函数就是 `shutil.copytree()`，它有一个 `ignore` 参数，可以将这个参数设置为一个函数。这个函数接受两个参数：一个根目录和一个文件列表(这个函数必须接受两个参数，即使它们根本没有被用到)，然后返回另一个文件名列表，`shutil.copytree()` 会忽略这个列表中的文件名。之后，这个函数在树中的每个文件夹被复制时都会被 `shutil.copytree()` 调用，参数为树中的当前目录和这个文件夹调用 `os.listdir()` 产生的文件列表。这对于忽略那些临时文件或归档文件，或这之后会被重新创建的文件是非常有用的。下面的这个简短示例展示了如何复制一个项目目录树，但忽略了任何已编译的 Python 文件(也就是那些扩展名为 `.pyc` 的文件)。

```
>>> def ignore_pyc(root, names):
...     return [name for name in names if name.endswith('.pyc')]
... 
```



```
>>> # now test that it works
>>> ignore_pyc('fred', ['1.py', '2.py', '2.pyc', '4.py', '5.pyc'])
['2.pyc', '5.pyc']
>>> sh.copytree('projdir', 'projbak', ignore=ignore_pyc)
```

在这个示例中，使用一个列表推导创建了一个忽略列表，但也可以只返回一个硬编码的文件名(例如，用来避免复制版本控制文件的 RCS)或者可以有一个更加复杂的涉及数据库查询或其他复杂处理的逻辑片段。测试标准模式(在本例中是`*.pyc`)的情况十分常见，所以 `shutil` 包含了一个名为 `shutil.ignore_patterns()` 的辅助函数。这个函数接受一个 `glob` 风格的模式列表作为参数并返回一个可以被用在 `shutil.copytree()` 中的函数。下面还是之前的示例，但是这次使用了 `shutil.ignore_patterns()`：

```
> > > sh.copytree('projdir', 'projbak', ignore=sh.ignore_patterns('*.pyc'))
```

记住，对于每个被复制的文件夹都会调用 `ignore` 函数，所以如果它非常复杂，`copytree()` 操作可能变得非常耗费资源并且很慢。

最后研究一下 `os` 中的 `os.path` 子模块。`os.path` 模块包含几个非常有用的测试函数和实用工具函数，在你使用前面已讨论过的高级函数时它们非常有用。最有用的函数主要用于创建路径、解构路径、扩展用户详情、测试路径存在性和获取关于文件属性的一些信息。

首先我们通过了解一些有用的测试函数来研究 `os.path`。如前所述，可以用 `os.stat()` 来获取关于文件的信息。`os.path` 提供了一些辅助函数，可以更加轻松地获取更多的常用特性。例如，可以使用 `os.path.getsize()` 来确定某个文件的大小，使用 `os.path.getmtime()` 获取文件的修改时间，使用 `os.path.getctime()` 获取文件的创建时间。也可以使用 `os.path.isfile()` 或 `os.path.isdir()` 来确定 `os.listdir()` 返回的是一个文件还是一个目录(甚至可以测试对你来说很重要的挂载点和链接)。所有这些函数接受一个名称作为参数并返回 `True` 或 `False`。这要比调用 `os.stat()` 然后使用索引和位掩码来提取详情信息更加轻松。

`os.path` 的另一个作用是处理路径。可以使用 `os.path.abspath()` 找出文件的完整路径。而如果它是一个链接，则可以用 `os.path.realpath()` 来获得真实文件的路径。在获得路径之后，可以将它分解为多个组成部分。Python 认为一个完整的文件路径看起来应该如下所示：

```
[<drive>]<path to folder><filename><extension>
```

使用 `os.path.splitdrive()`，可以读取驱动盘符(如果是在 Windows 上，否则它是空的)。`os.path.dirname()` 可以找到文件夹，而 `os.path.basename()` 可以获得文件名(包括扩展名)。你甚至可以使用 `os.path.split()` 一次就获得文件夹路径和文件名。通常来说这就足够用了。但是如果有必要，则可以使用 `os.path.splitext()` 进一步将文件名分成它的扩展名和核心名。这样获得的扩展名包含了点符号，比如 `myfile.exe` 返回 `myfile` 和 `.exe`。

通常，在检查和使用各种路径组成部分之后，你想要重新组合路径或者甚至从头开始创建一个路径。`os.path` 为此提供了另一个便利函数 `os.path.join()`，它接受数个元素作为参数并使用定义在常量 `os.sep` 中的当前操作系统的路径分隔符将它们组合成一个单独的字符串。这很重要，因为路径格式在不同操作系统之间会有显著的不同。自从基于 UNIX 内核

的 MacOS X 出现后,事情已经变得容易些了。因为除了它本身的原生\风格,Windows 通常也接受 UNIX 风格的/分隔符。但是如果计划让你的脚本在多个计算机类型上运行,那么使用 `os.path.join()` 创建文件路径要更加安全些。

可以在你的测试文件和文件夹中看到该操作的执行。

试一试: 使用路径

在这个“试一试”中,使用 `os.path` 函数来确定各种文件的状态。完成下面的步骤:

- (1) 切换到之前创建的根目录。暂时待在操作系统中。
- (2) 使用操作系统,将 `fileA.txt` 复制到子目录 `subdir` 中的 `fileC.txt`。
- (3) 现在在 `subdir` 中创建一个指向 `fileB.txt` 的名为 `fileD.txt` 的符号链接(如果在 Windows 上,可能不能创建符号链接,所以需要忽略关于 `fileD.txt` 的步骤)。该操作的 UNIX 命令如下所示:

```
$ ln -s fileB.txt subdir/fileD.txt
```

- (4) 启动 Python 并输入下面的代码:

```
>>> import os
>>> from os import path
>>> os.listdir('.')
['fileA.txt', 'fileB.txt', 'ls.txt', 'subdir', 'test.py']
>>> path.getsize('fileA.txt')
257
>>> path.getctime('fileA.txt')
1389109373.1044922
>>> path.getsize('subdir/fileC.txt')
257
>>> path.getctime('subdir/fileC.txt')
1389274706.736207
>>> path.abspath('fileB.txt')
'/home/agauld/book/root/fileB.txt'
>>> path.abspath('subdir/fileD.txt') # Not Windows
'/home/agauld/book/root/subdir/fileD.txt'
>>> path.islink('fileB.txt') # Not windows
False
>>> path.islink('subdir/fileD.txt') # Not windows
True
>>> path.realpath('subdir/fileD.txt') # Not windows
'/home/agauld/book/root/subdir/fileB.txt'
>>> folder, filename = path.split(path.abspath('subdir/fileB.txt'))
>>> print (folder, filename)
/home/agauld/book/root/subdir fileB.txt
>>> path.join(folder,filename)
'/home/agauld/book/root/subdir/fileB.txt'
>>> path.splitext(filename)
('fileB', '.txt')
```

示例说明

一开始，你导入了需要的模块并且检查当前文件夹的内容，这只是为了验证你在正确的地方。然后对比了 `fileA.txt` 和 `subdir/fileC.txt`(回想一下，它是 `fileA.txt` 的副本)的大小和创建时间。不出所料，它们的大小相同但创建时间不同。

然后你查看了 `fileB.txt` 和与它有链接关系的 `fileD.txt`。这一次，你首先查看了每个文件的绝对路径。不出所料，它们都展示了各自的位置。然后你测试了两个文件，查看它们是否是链接，当然，`fileB.txt` 不是链接而 `fileD.txt` 是链接。之后你寻找 `fileD.txt` 的真实路径。它的原件实际上是 `fileB.txt`。

最后，你了解了一些路径操纵的方法。使用 `os.path.split()`将 `fileB.txt` 的路径分解为文件夹和文件部分。使用 `os.path.join()`将它们连接到一起，然后将文件名分解为核心名和扩展名。注意，点还留在扩展名的前面。

2.1.7 探索目录树深度

一个常见的自动化操作是在一个给定的位置启动并对这个位置下的文件系统中的一个文件(或文件类型)都应用一个特定的行为。这通常被称为遍历目录树。而 `os` 模块中有一个强大且灵活的函数 `os.walk()`，它可以帮助你实现这一点。由于它不是那种最简单易用的函数，所以你需要在本节中了解它的主要特性。

本节将介绍使用 `os.walk()`在一个给定的目录树或子树中查找某个特定文件的示例。然后将创建一个可以用在程序中的包含了 `findfile()`函数的新模块。基于此，可以构成一组用来处理目录树的函数。

首先，需要创建一个在根目录下的文件夹架构的测试环境(可以通过解压下载网站中 `Chapter2.zip` 主文件内的 `TreeRoot.zip` 来生成该结构，或者可以使用操作系统工具手动生成它)。每个文件夹包含一些文件，而其中一个文件夹包含着你想要找的文件 `target.txt`。该结构如下：

```
TreeRoot
  FA.txt
  FB.txt
  D1
    FC.txt
    D1-1
      FF.txt
  D2
    FD.txt
  D3
    FE.txt
    D3-1
      target.txt
```

`os.walk()`函数结构以一个开始点作为参数，并返回一个生成元组的生成器，其中这个元组包含三个成员(有时称为三元组(3-tuple 或 triplet))。这三个成员为根目录、在当前根目

录下的目录列表和根目录中的当前文件列表。如果查看一下已经创建的架构，顶层元组应该如下所示：

```
( 'TreeRoot', ['D1', 'D2', 'D3'], ['FA.txt', 'FB.txt'] )
```

可以轻松地在交互式提示符中写一个 for 循环来查看它：

```
>>> import os
>>> for t in os.walk('TreeRoot'):
...     print (t)
...
('TreeRoot', ['D1', 'D2', 'D3'], ['FA.txt', 'FB.txt'])
('TreeRoot/D1', ['D1-1'], ['FC.txt'])
('TreeRoot/D1/D1-1', [], ['FF.txt'])
('TreeRoot/D2', [], ['FD.txt'])
('TreeRoot/D3', ['D3-1'], ['FE.txt'])
('TreeRoot/D3/D3-1', [], ['target.txt'])
```

上面的代码清晰地展示了 `os.walk()` 输入的路径，从顶层的第一个目录开始向下深度优先遍历它。也展示了如何从一个文件列表中取出一个文件并且通过组合名称和元组中的根目录值来构建它的完整路径。

通过编写函数来使用正则表达式并返回一个列表，可以创建一个比之前看到的 `glob.glob()` 强大得多(但也更慢)的函数。

试一试：构建文件查找器(file_tree.py)

在这个“试一试”中，你将构建并测试一个模块，其中包含一个使用 `os.walk()` 函数作为基础的文件查找函数。为此，需要完成下面的步骤：

(1) 如果还没有用于测试的目录结构，就如之前所述的那样创建一个(或者从 zip 文件中加载它)。

(2) 切换到保存 Python 模块的项目目录。

(3) 打开你最喜欢的文本编辑器，并输入下面的代码(或者从 zip 文件中加载它)：

```
# file_tree.py module containing functions to assist
# in working with directory hierarchies.
# Based on the os.walk() function.
```

```
import os, re
import os.path as path
```

```
def find_files(pattern, base='.'):
    """Finds files under base based on pattern

    Walks the file system starting at base and
    returns a list of filenames matching pattern"""

    regex = re.compile(pattern)
    matches = []
```

```

for root, dirs, files in os.walk(base):
    for f in files:
        if regex.match(f):
            matches.append( path.join(root,f) )
return matches

```

(4) 将文件保存为 file_tree.py。

(5) 切换到根目录(也就是 TreeRoot 的上一层目录)。

(6) 启动 Python 解释器, 输入下面的代码来测试新函数:

```

>>> import file_tree
>>> help(file_tree)
Help on module file_tree:

```

NAME

file_tree

DESCRIPTION

```

# file_tree.py module containing functions to assist in
# working with directory hierarchies.
# Based on the os.walk() function.

```

FUNCTIONS

```

find_files(pattern, base='.')
    Finds files under base based on pattern

Walks the file system starting at base and
returns a list of filenames matching pattern

```

FILE

/cygdrive/d/PythonCode/Chapter2/file_tree.py

(7) 输入 q, 退出帮助屏幕:

```

>>> file_tree.find_files('target.txt','TreeRoot')
['TreeRoot/D3/D3-1/target.txt']
>>> file_tree.find_files('F.*','TreeRoot')
['TreeRoot/FA.txt', 'TreeRoot/FB.txt', 'TreeRoot/D1/FC.txt',
'TreeRoot/D1/D1-1/FF.txt', 'TreeRoot/D2/FD.txt', 'TreeRoot/D3/FE.txt']
>>> file_tree.find_files('.*\\.txt','TreeRoot')
['TreeRoot/FA.txt', 'TreeRoot/FB.txt', 'TreeRoot/D1/FC.txt',
'TreeRoot/D1/D1-1/FF.txt', 'TreeRoot/D2/FD.txt', 'TreeRoot/D3/FE.txt',
'TreeRoot/D3/D3-1/target.txt']
>>> file_tree.find_files('D.*','TreeRoot')
[]

```

示例说明

这个函数接受一个正则表达式作为参数 pattern, 并且为了效率而编译了它。然后该函数以正常的方式使用基础参数值在目录树的每一层调用 os.walk(), 检查每个找到的文件是

否符合输入的模式。如果找到了一个匹配，就生成完整路径并把它加入到结果列表中。一旦目录树中的所有文件都已经过测试，函数就返回找到的文件的列表。

通过导入模块并在它上面运行 `help()` 函数，你测试了代码。你看到了用来描述该函数使用方式的注释和文档字符串。

在退出帮助之后，你通过搜索文件名 `target.txt` 测试了 `find_files()`，并且得到了结果 `TreeRoot/D3/D3-1/target.txt`。然后你实验了一些正则表达式(请注意 `glob()` 通配符语法和正则表达式形式之间的差别)。F.*是指任何以 F 开头后面跟着 0 个或多个字符的文件。`.*\txt` 指的是任何字符序列后面跟着一个点和三个字符 `txt`。

最后，你试图使用模式来匹配目录名，但是得到了一个空列表。这是在意料之中的，因为函数只检查文件名，它并不查看 `dirs` 元组元素中的名称。

你已经看到了在 Python 中如何使用操作系统，下一节将介绍在 Python 中如何使用日期和时间。

2.2 使用日期和时间

一个最常用的脚本任务特性就是使用日期和时间。使用它们可以确定那些比某一日期更久或在某一个范围内的文件，或者可以设置一个进程在某一时间或间隔内执行。你可能需要去对比数据中的日期和时间来选择文件内容中合适的子集。在许多场景中，读取日期和时间并对比它们的值是必要的。

遗憾的是，日期和时间并不是清晰地被定义为类似整数或浮点数的值。它们更倾向于被存储为多种格式的字符串。例如，`2016-02-07`、`02/07/2016` 和 `07/02/2016` 全都是 2016 年 2 月 7 日可能的表达方式。情况可能会变得更复杂，因为还可能使用月份和天的名称缩写，例如 `Jan`、`Feb`、`Mon` 或 `Tue` 等。此外，年可能被缩写为两个数字而分隔符可能是任意数量的字符，这又会大大地增加复杂性。如何才能可靠地从一个给定的字符串读取日期值呢？时间值也很复杂，尤其是当必须考虑时区和夏令时规则时。幸运的是，Python 提供了几个模块来帮助你解决这些问题。最基本的是 `time` 模块，`datetime` 模块对它进行了扩展，而 `calendar` 模块对一些任务特别适用。

2.2.1 使用 time 模块

`time` 模块以两种不同的格式存储时间(包括日期)。第一种是自新纪元(epoch)以来的秒数，新纪元在历史上是一个固定的日期。对于基于 UNIX 的系统，它是 1970 年 1 月 1 日(注意，这也是另一个日期表达方式么？)。另一种表达方式是一个包含多个域的元组，用来表示一个日期或时间的各个部分：年、月、日、小时、分钟、秒等。这些细节都可以在 `time` 模块文档中找到，但你需要记住使用的是哪种基本格式。`time` 模块包含了多个转换函数在它们之间切换。

有两个非常重要的函数与刚才讨论的大部分话题有关。这两个函数为 `strptime()`(`p` 代表解析(parse))和 `strftime()`(`f` 代表格式化(format))。它们从字符串读取时间并将时间写成字符串。使用它们的秘密就在于格式化字符串。这个字符串告诉函数如何将字符串值映射到时间和如何从时间值映射为字符串。格式化字符串使用%标记来表示一个域,使用一组字符代码来表示这个域应该包含什么。比如,%Y 表示一个四位的年,而%y 表示一个两位的年。%m 表示一个两位的月,而%B 表示月份的全称(会考虑本地语言设置)。`time` 模块文档中的 `strftime()` 部分提供了完整的列表。



注意: 需要注意的一个时间字符串函数是 `strftime()` 函数,该函数接受格式化字符串作为它的第一个参数,要转换的元组作为它的第二个参数。`strptime()` 函数接受输入字符串作为它的第一个参数,格式化字符串作为它的第二个参数。

掌握这些函数最简单的方法就是在 Python 命令提示符中使用它们。你现在就可以试试。

试一试：格式化日期和时间字符串

在这个“试一试”中,你会看到如何使用 `strptime()` 和 `strftime()` 函数。要完成下面的步骤,需要你随时试一试其他变种来巩固理解。

(1) 启动 Python 解释器,输入下面的代码:

```
>>> import time as t
>>> now = t.time() # Note: current time will be a different value each time
>>> now
1394536692.958137
>>> gmt = t.gmtime(now)
>>> gmt
time.struct_time(tm_year=2014, tm_mon=3, tm_mday=11, tm_hour=11,
tm_min=18, tm_sec=12, tm_wday=1, tm_yday=70, tm_isdst=0)
>>>
```

注意:当前时间是时时变化的。

(2) 看一下当前时间的两种格式:从新纪元以来的秒数和 `gmt`(表示 GMT 或 UTC 时间的时间元组)。接下来,你使用元组版本来使用 `strftime()`。

(3) 输入下面的代码:

```
>>> t.strftime("The date is: %Y-%m-%d", gmt)
'The date is: 2014-03-11'
>>> t.strftime("The date is: %b %d, %Y", gmt)
'The date is: Mar 11, 2014'.s
>>> t.strftime("The time is: %H:%M:%S", gmt)
```

```
'The time is: 11:18:12'
>>> t.strftime("It is now %I %M%p",gmt)
'It is now 11 18AM'
>>> t.strftime("The local time format is: %X", gmt)
'The local time format is: 11:18:12'
>>> t.strftime("The local date format is: %x", gmt)
'The local date format is: 03/11/14'
>>>
```

(4) 输入下面的代码，看看使用 `strptime()` 的逆操作。

```
>>> dt = t.strptime("Saturday, March 8, 2014", "%A, %B %d, %Y")
>>> dt
time.struct_time(tm_year=2014, tm_mon=3, tm_mday=8, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=5, tm_yday=67, tm_isdst=-1)
>>> dt = t.strptime("Saturday, March 8th, 2014", "%A, %B %d, %Y")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.2/_strptime.py", line 482, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/usr/lib/python3.2/_strptime.py", line 337, in _strptime
    (data_string, format))
ValueError: time data 'Saturday, March 8th, 2014' does not match
format '%A, %B %d, %Y'
>>> dt = t.strptime("Saturday, March 8th, 2014", "%A, %B %dth, %Y")
>>>
```

(5) 注意，格式化字符串必须与输入字符串严格匹配。在天后面使用一个 `th` 后缀让 `strptime()` 函数完全令人困惑了。

(6) 再多试试一些示例：

```
>>> t.strptime("2014 - 01 - 01", "%Y - %m - %d")
time.struct_time(tm_year=2014, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=2, tm_yday=1, tm_isdst=-1)
>>> t.strptime("2014 - 01 - 01T15:05:45", "%Y - %m - %dT%H:%M:%S")
time.struct_time(tm_year=2014, tm_mon=1, tm_mday=1, tm_hour=15,
tm_min=5, tm_sec=45, tm_wday=2, tm_yday=1, tm_isdst=-1)
>>> t.asctime(gmt)
'Tue Mar 11 11:18:12 2014'
>>> t.mktime(dt)
1394236800.0
>>>
```

示例说明

首先，你导入了 `time` 模块，而为了省去一些输入，你为它使用了别名 `t`。然后使用 `time.time()` 函数来获得当前时间。这可以反映出你电脑的当前时间，所以得到的值与显示的是不同的。实际上，每一次调用函数得到的值都应该是不同的。`now` 变量包含了表示为自新纪元以来的秒数的时间。注意它是一个十进制浮点数值。十进制小数点后面的部分依赖

于你的操作系统和有关它准确性的计算机时钟，所以它可能并不是像它表现的那样精确。秒表示法对于简单的时间运算是非常有用的，例如，为代码中两个事件之间的持续时间计时(对于更加复杂的运算，可以考虑使用将在下一节中讨论的 `datetime` 模块)。

接下来，你将时间从秒转换为一个时间元组，它是你为了使用字符串格式化函数需要的格式。元组表示法能让你确定 `now` 的值确实存储着当前的日期和时间。

然后，使用 `strftime()` 用各种格式打印存储的时间。注意，格式化字符串中可以包含任何字符串文本，不仅仅是特殊的格式化字符。

对于之后的一组指令，你使用 `strptime()` 函数将一个字符串转换成一个日期元组。第一个示例使用了一个格式良好的字符串将值存储在一个名为 `dt` 的变量中。接下来的示例使用了一个格式较糟糕的字符串，因为它在日的值后面有一个 `th` 后缀。`strptime()` 解析器不能够处理这种格式，所以需要将这个后缀添加到字符串值中。当读取这种格式的字符串时，会显得很笨重。而且你可能需要使用一个 `try/except` 格式加上一个后缀组合(`st`、`nd`、`rd`、`th`)来使它正确。然后你又尝试了一些其他格式，包括其他时间。

最后，你看到了两个新的函数：`time.asctime()` 是一个用来不管本地设置而使用标准格式打印时间元组的便利函数；`time.mktime()` 将一个时间元组转换成一种秒表示形式。

`time` 模块包含了其他几个用来管理时区和获取系统时钟信息的函数，也可以判断在当前计算机上夏令时是否有效。

最后，`time` 模块包含了一个 `sleep()` 函数，它可以将你的程序暂停给定量的秒数。这个函数在脚本中常常很有用，例如，你正在使用后台进程执行一个任务而这个任务可能需要一些时间。它在你轮询资源时也很有用，例如，等待数据到达的网络连接。可以使用一秒的几分之一，但是你应该意识到这个计时仅仅是近似的，因为会存在操作系统进程调度开销等。

2.2.2 datetime 模块介绍

`datetime` 模块包含了几个对象和方法，用来表示绝对日期和时间以及相对日期和时间。相关的值被用来计算时间之间的差别，从而避免你必须在基于秒的数值上做混乱的运算，例如，除以 60 和 24 等。`time` 函数和 `datetime` 对象之间有一些重叠的地方。通常来说，如果在作比较或基于时间的运算时，你应该使用 `datetime` 模块而不是 `time` 模块。如果在同一段代码中两个都使用的话，使用最基本的 `import` 风格可以保证没有命名冲突发生。

`datetime` 模块使用的主要类有 `date`、`time` 和 `datetime`。它们的名称标示着它们的范围。`datetime` 和 `time` 对象可以通过将它们的时间区特性设置为 `timezone` 对象来考虑时区的作用。如果要做复杂的时间处理，你可能需要子类化 `timezone` 类来提供任何需要的复杂算法。在本书中，你只会用到模块中的基本对象。另一个可能也是最有用的对象类型是 `timedelta` 类。它处理时间计算或相对时期，比如一年或一个月的时间间隔。`datetime` 模块使用 `timedelta` 对象支持许多基于时间的计算，包括加法、减法、时间差和一个数的乘法，以及甚至各种形式的除法。

可以通过传入年、月和日的值来初始化 `date` 对象，这几个值是必须传入的。可以通过传入小时、分钟和秒的值来初始化 `time` 对象，这几个值是可选的，默认都是 0。你不会惊讶于 `datetime` 对象使用了所有年、月、日、小时、分钟和秒。一些有用的类方法返回基于对象参数的实例。一个示例是 `date.today()` 方法，它返回当天的日期。还有 `date.fromtimestamp()` 方法，它接受一个秒的时间值作为参数。在 `date` 被创建之后，可以通过很多特性和方法来提取数据。`date` 类包含一个类似于 `time` 模块的 `strftime()` 方法(但是没有对应的 `strptime()`，为此，你必须查看 `datetime` 对象)。

`time` 对象在概念上很相似，但是正如之前所述，它拥有考虑时区的能力，包括夏令时信息。`time` 对象与 `date` 对象一样只支持 `strftime()` 方法。

`datetime` 对象是 `date` 对象和 `time` 对象的结合，它支持两个对象方法的组合。`datetime` 也添加一些它自己的其他方法，包括初始化为当前日期和时间的 `now()` 类方法，和接受 `date` 和 `time` 对象作为参数返回一个组合的拥有同样值的 `datetime` 对象的 `combine()` 类方法。可以使用 `datetime` 和 `timedelta` 对象的组合做一些基本的算术运算。后者既可以作为参数，在某些情况下也可以作为结果。`datetime` 对象也支持 `strftime()` 和 `strptime()` 方法，它们与之前在 `time` 模块中描述的方法的工作原理是一样的。

在本章后面的“试一试：使用 `ElementTree` 解析 XML(`toohireEt1.py` 和 `toohireET.py`)”中，你会使用 `datetime` 对象，它是较大示例的一部分。

2.2.3 calendar 模块介绍

`calendar` 模块是 Python 标准库中最简单的基于时间的模块。本质上，它为一个给定的年生成一个日历。这个日历是 `calendar.Calendar` 类实例。它有很多支持方法，允许你遍历给定月的每一天或者在脚本中生成各式各样的文本字符串，这些字符串在表示脚本中的用户消息时十分有用。日历可以被格式化成纯文本文件或在 HTML 中被格式化。

`calendar` 可能是三个被讨论的模块中最少被用到的模块。但是它有很多有用的特性，这些特性在其他地方没有而且如果重新实现的话会很费时间。在这些特性中有一些是实用工具函数，例如 `isleap()` 和 `time.gmtime()`。`isleap()` 报告一个特定的年是否是一个闰年。`time.gmtime()` 将一个 `time.gmtime()` 元组转换成秒(为什么它在 `calendar` 模块中而不在 `time` 模块中是一个谜)。

最后，两个打印函数 `prcal()` 和 `prmonth()` 分别接受一个年和一个月/年/月组合作为参数，并在 `stdout` 上显示它们的输出。当提示用户选择一个日期时，这些函数就非常有用。

还有一些可用的第三方模块。它们试图通过把所有标准库中的所有函数组合成一个单独的更加易用的模块来简化 Python 中日期和时间的处理。一些示例包括 `arrow` 和 `delorean`。但是通过因特网搜索，还可以找到其他的模块。

在下一节中，你会看到在 Python 中如何读取和写入一些常用的数据文件格式。

2.3 处理常见的文件格式

在编写脚本控制一些应用或实用工具时，通常会使用文件作为应用之间的数据传输机制。遗憾的是，一个程序的输出格式可能不是下一个应用读取的正确格式。在这一点上，脚本本身必须把输出文件转换成下一个应用能够读取的合适形式。大多数应用生成和使用一些标准格式的变体，比如 CSV(逗号分隔值)、HTML(超文本标记语言)、XML(可扩展标记语言)、Windows INI(以文件扩展名命名)以及最新的 JSON(基于 JavaScript 语言的轻量级的数据交换格式)。现在你将会看到 Python 标准库是如何支持这些格式的(JSON 主要在第 5 章的 5.3 节讲述，因为它更常被用于 Web 应用中)。相比使用标准 Python 文本处理工具，比如字符串方法或正则表达式，来读取和写入数据，这些模块让它变得更加易用。

2.3.1 使用逗号分隔的数值

逗号分隔的数值(CSV)格式已经出现多年了。它的名称其实有点用词不当，因为尽管逗号是最常用的分隔符，但是术语 CSV 经常被应用于使用制表符或管道符(|)或者甚至几乎任何其他类型的符号作为分隔符的文件。乍一看，使用内置的字符串 `split()` 方法从这样的文件中解析看起来很轻松。但问题是，格式并不是绝对的标准，而且不同的文件有不同的表示域的方式，域中可能包含着分隔符。同时，多行数据有时可能被分割为文件中的多个物理行。为了更加轻松地处理这种多样性，Python 在它的标准库中包含了 `csv` 模块。

`csv` 模块提供了两种机制来读取 CSV 文件。最简单的机制只是读取每行到一个元组中，然后程序员需要记录元组中的每个位置代表什么意思。第二种方法读取数据到一个字典中，通常使用文件的第一行作为字典的键。这是一个非常灵活的机制，因为它不需要打乱现有的代码就可以适应文件格式的改变(比如添加新键)。



注意：标准 `csv` 模块的一个大问题是它不能处理 Unicode 输入。如果需要处理包含 Unicode 字符的文件，你应该试试第三方模块，比如 `unicodcsv`。

模块默认使用 Microsoft Excel 的 CSV 格式，但是也可以定义自己的格式。这只需要做一点额外工作。在这一章，你只会处理 Excel 格式。

接下来的示例基于简单的电子表格文件 `toolhire.xlsx`，如图 2-1 所示。这个电子表格描述了一个小型工具雇佣设施。它是由一些朋友创建的，用来追踪谁从谁那借了什么。

toolhire.xlsx							
	A	B	C	D	E	F	G
1	ItemID	Name	Description	Owner	Borrower	DateLent	DateReturned
2	1	LawnMower	Small Hover mower	Fred	Joe	4/1/2012	4/26/2012
3	2	LawnMower	Ride-on mower	Mike	Anne	9/5/2012	1/5/2013
4	3	Bike	BMX bike	Joe	Rob	7/3/2013	7/22/2013
5	4	Drill	Heavy duty hammer	Rob	Fred	11/19/2013	11/29/2013
6	5	Scarifier	Quality, stainless steel	Anne	Mike	12/5/2013	
7	6	Sprinkler	Cheap but effective	Fred			
8							

图 2-1 工具雇佣数据表

数据以 CSV 格式保存在文件 `toolhire.csv` 中。文件中的原始数据看起来如下所示：

```
ItemID,Name,Description,Owner,Borrower,DateLent,DateReturned
1,LawnMower,Small Hover mower,Fred,Joe,4/1/2012,4/26/2012
2,LawnMower,Ride-on mower,Mike,Anne,9/5/2012,1/5/2013
3,Bike,BMX bike,Joe,Rob,7/3/2013,7/22/2013
4,Drill,Heavy duty hammer,Rob,Fred,11/19/2013,11/29/2013
5,Scarifier,"Quality, stainless steel",Anne,Mike,12/5/2013,
6,Sprinkler,Cheap but effective,Fred,,,
```

这是一个非常普通的文件，但是包含了之前描述的复杂性。注意，Anne 的松土机的描述有双引号，因为它含有一个逗号。

在导入模块后可以按如下方式将文件读入到一个元组列表中：

```
>>> import csv
>>> with open('toolhire.csv') as th:
...     toolreader = csv.reader(th)
...     print(list(toolreader))
...
[['ItemID', 'Name', 'Description', 'Owner', 'Borrower',
'DateLent', 'DateReturned'],
['1', 'LawnMower', 'Small Hover mower', 'Fred', 'Joe', '4/1/2012', '4/26/2012'],
['2', 'LawnMower', 'Ride-on mower', 'Mike', 'Anne', '9/5/2012', '1/5/2013'],
['3', 'Bike', 'BMX bike', 'Joe', 'Rob', '7/3/2013', '7/22/2013'], ['4', 'Drill',
'Heavy duty hammer', 'Rob', 'Fred', '11/19/2013', '11/29/2013'], ['5',
'Scarifier', 'Quality, stainless steel', 'Anne', 'Mike', '12/5/2013', ''],
['6', 'Sprinkler', 'Cheap but effective', 'Fred', '', '', '']]
>>>
```

注意，Anne 的松土机描述不再有双引号了，但是还包含初始的逗号。理解了这是如何实现的，就是 csv 模块给你程序带来的价值。可以在调用 `open()` 的文件和 `csv.reader` 对象创建过程中应用许多选项。这个示例展示了最小的选项集合。



注意：reader 对象并不仅限于文件。它也可以把一个字符串列表作为它的输入。如果使用 `subprocess` 在 `stdout` 上产生 CSV 格式数据，这是一个强大的工具。

向一个 CSV 文件写入数据也同样简单。在这个示例中，你为 `toolhire.xlsx` 电子表格创建了一个新的数据页，它列出了各种可用的工具，以及有关它们何时可用、状态和原价等细节。你将这些数据存入到名为 `tooldesc.csv` 的 CSV 文件，可以将该文件作为一个新工作表加载到 Excel 中。

代码如下：

```
>>> import csv
>>> items = [
...     ['1','Lawnmower', 'Small Hover mower', 'Fred','$150','Excellent',
...     '2012-01-05'],
...     ['2','Lawnmower','Ride-on mower','Mike','$370','Fair','2012-04-01'],
...     ['3','Bike','BMX bike','Joe','$200','Good','2013-03-22'],
...     ['4','Drill','Heavy duty hammer','Rob','$100','Good','2013-10-28'],
...     ['5','Scarifier','Quality, stainless steel','Anne','$200','2013-09-14'],
...     ['6','Sprinkler','Cheap but effective','Fred','$80','2014-01-06']
... ]
>>> with open('tooldesc.csv','w', newline='') as tooldata:
...     toolwriter = csv.writer(tooldata)
...     for item in items:
...         toolwriter.writerow(item)
...
44
39
33
34
34
33
>>>
```

如你所见，`writer.writerow()`方法返回写入文件的字符数。基本上可以忽略这些！输出文件如下：

```
1,Lawnmower,Small Hover mower,Fred,$150,Excellent,2012-01-05
2,Lawnmower,Ride-on mower,Mike,$370,Fair,2012-04-01
3,Bike,BMX bike,Joe,$200,Good,2013-03-22
4,Drill,Heavy duty hammer,Rob,$100,Good,2013-10-28
5,Scarifier,"Quality, stainless steel",Anne,$200,2013-09-14
6,Sprinkler,Cheap but effective,Fred,$80,2014-01-06
```

注意，松土机的描述又有双引号了，而且日期域也被精确地写成它本来的样子。如果想要获得与 Excel 同样格式的日期，在写数据之前就需要完成那个操作。在使用 CSV 文件作为在不同应用间的传输媒介时，你会发现这是非常典型的不一致。可以使用 `datetime` 模块来转换日期格式。`datetime` 包含 `datetime.strptime()` 函数和 `datetime.strftime()` 函数。`datetime.strptime()` 函数可以把一个输入字符串解析成一个 `datetime` 对象。`datetime.strftime()` 函数可以把 `datetime` 对象以你想要的格式写出。现在试试吧。

试一试：重新格式化数据并写入到 CSV 文件(change_date.py)

在这个“试一试”中，你会从 `tooldesc.csv` 文件读取工具列表，然后从列表中提取每个日期域，转换成日期格式，最后将这个数据格式写回到 CSV 文件中。完成下面的步骤：

- (1) 切换到存储 CSV 文件的目录，或者创建一个新目录并把 CSV 文件从 zip 文件中复制过来。
- (2) 打开你喜欢的 IDE 或编辑器。输入下面代码并保存为 `change_date.py` (或者从 Chapter2.zip 下载文件的 ToolHire 文件夹中加载它)：

```
import csv
from datetime import datetime

def convertDate(item):
    theDate = item[-1]
    dateObj = datetime.strptime(theDate, '%Y-%m-%d')
    dateStr = datetime.strftime(dateObj, '%m/%d/%Y')
    item[-1] = dateStr
    return item

with open('tooldesc.csv') as td:
    rdr = csv.reader(td)
    items = list(rdr)

items = [convertDate(item) for item in items]
with open('tooldesc2.csv', 'w', newline='') as td:
    wrt = csv.writer(td)
    for item in items:
        wrt.writerow(item)
```

- (3) 运行脚本。查看新的 `tooldesc2.csv` 文件是否已经创建。
- (4) 在文本编辑器(如 Notepad)中打开新文件 `tooldesc2.csv`。
- (5) 确定它有新的如下日期格式：

```
1,Lawnmower,Small Hover mower,Fred,$150,Excellent,01/05/2012
2,Lawnmower,Ride-on mower,Mike,$370,Fair,04/01/2012
3,Bike,BMX bike,Joe,$200,Good,03/22/2013
4,Drill,Heavy duty hammer,Rob,$100,Good,10/28/2013
5,Scarifier,"Quality, stainless steel",Anne,$200,09/14/2013
6,Sprinkler,Cheap but effective,Fred,$80,01/06/2014
```

示例说明

首先，你导入了所需要的 `csv` 模块和 `datetime` 模块中的 `datetime` 类。然后创建了一个函数来转换日期。该函数首先从记录中提取日期域，然后使用 `datetime.strptime()` 函数来解析日期域。格式化字符串 `("'%Y-%m-%d'")` 告诉它选择一个四位数字年(`%Y`)后跟一个连字符、一个两位数字月份(`%m`)、一个连字符和一个两位数字的天(`%d`)。这会产生一个日期对象。然后使用 `datetime.strftime()` 函数通过重新排列域并使用斜线(`/`)作为分隔符把那个日期对象

格式化需要的格式。

(注意, `datetime` 版本的 `strptime()` 和 `strftime()` 不存在像 `time` 模块版本的参数顺序不一致的现象)。

最后, 你使用新字符串替换了原始的日期域, 并返回被修改后的记录。

脚本的主要代码打开了原始的 CSV 文件, 读取记录并把它们保存在名为 `items` 的列表中。然后, 你使用一个列表推导替换 `items`。这个列表推导对 `items` 中的每个记录都调用了 `convertDate()` 函数。

最后, 你把修改后的 `items` 列表写入到名为 `tooldesc2.csv` 的新 CSV 文件中。

到目前为止, 已经使用了 `csv` 模块中基本的 `reader` 和 `writer` 组件来操作数据项列表。回想一下, 之前 `csv` 也支持一个基于字典的方法。现在你使用这种方法来访问初始的 `toolhire.csv` 文件。如果再看看 CSV 文件的内容, 你会注意到第一行是用来描述列的标题列表。`csv` 模块可以通过把标题作为一个字典的键来利用它们。这让访问单独的域变得更加可靠, 因为不再需要依赖域在文件中的数字位置。

它工作的方式与之前的代码很相似, 但不是使用 `csv.reader` 对象, 而是使用 `csv.DictReader` 对象。代码如下:

```
>>> with open('toolhire.csv') as th:
...     rdr = csv.DictReader(th)
...     for item in rdr:
...         print(item)
...
{'DateReturned': '4/26/2012', 'Description': 'Small Hover mower',
'Owner': 'Fred', 'ItemID': '1', 'DateLent': '4/1/2012',
'Name': 'LawnMower', 'Borrower': 'Joe'}
{'DateReturned': '1/5/2013', 'Description': 'Ride-on mower',
'Owner': 'Mike', 'ItemID': '2', 'DateLent': '9/5/2012',
'Name': 'LawnMower', 'Borrower': 'Anne'}
{'DateReturned': '7/22/2013', 'Description': 'BMX bike',
'Owner': 'Joe', 'ItemID': '3', 'DateLent': '7/3/2013',
'Name': 'Bike', 'Borrower': 'Rob'}
{'DateReturned': '11/29/2013', 'Description': 'Heavy duty hammer',
'Owner': 'Rob', 'ItemID': '4', 'DateLent': '11/19/2013',
'Name': 'Drill', 'Borrower': 'Fred'}
{'DateReturned': '', 'Description': 'Quality, stainless steel',
'Owner': 'Anne', 'ItemID': '5', 'DateLent': '12/5/2013',
'Name': 'Scarifier', 'Borrower': 'Mike'}
{'DateReturned': '', 'Description': 'Cheap but effective',
'Owner': 'Fred', 'ItemID': '6', 'DateLent': '',
'Name': 'Sprinkler', 'Borrower': ''}
>>>
```

注意, 正如往常的字典一样, 域并不是初始的顺序, 它们根据第一行的标题来索引。可以看到, 就像之前一样, 松土机的描述去掉了双引号但是保留了逗号。

如果把它们存储在一个变量中而不打印，则可以使用列表推导对数据做一些有意思的分析。比如，查看所有 Fred 拥有的东西，如下所示：

```
>>> with open('toolhire.csv') as th:
...     rdr = csv.DictReader(th)
...     items = [item for item in rdr]
...
>>> [item['Name'] for item in items if item['Owner'] == 'Fred']
['LawnMower', 'Sprinkler']
>>>
```

可以使用基础的 reader 和它的列表来做同样的事情，但是你可能需要使用数字索引，这样可读性变得更差。比如，使用之前的列表进行列表推导看起来会是这样：

```
>>> [item[1] for item in toolList if item[3] == 'Fred']
['LawnMower', 'Sprinkler']
```

你的返回值或选择标准都不是很清晰。同时，如果文件格式改变了，你就需要改变代码中所有的索引值。

有一个很适合的 DictWriter 对象可以将字典写入到 CSV 文件。可以在下一个“试一试”中使用它。

即使你的 CSV 文件不包含标题，也可以使用 DictReader。例如，你在前面的“试一试”中创建的 tooldesc2.csv 文件就没有标题行。可以把它读入 DictReader 然后用 DictWriter 将它写出来。这可以弥补这个不足。技巧就是提供标题作为 DictReader 构造函数的参数。现在试一试吧。

试一试：将标题行添加到 CSV 文件(add_labels.py)

在这个“试一试”中，为 tooldesc2.csv 文件定义了一组标题，然后打开文件把它读入到一个 DictReader 对象。然后使用 DictWriter 将数据写到新文件。它会自动地插入一个标题行。请完成下面的步骤：

- (1) 切换到项目文件夹并打开 IDE 或编辑器。
- (2) 输入下面代码并把它保存为 add_labels.py(或者从 zip 文件加载它)：

```
import csv

fields = ['ItemID', 'Name', 'Description', 'Owner',
          'Price', 'Condition', 'DateRegistered']

with open('tooldesc2.csv') as td_in:
    rdr = csv.DictReader(td_in, fieldnames = fields)
    items = [item for item in rdr]

with open('tooldesc3.csv', 'w', newline='') as td_out:
    wrt = csv.DictWriter(td_out, fieldnames=fields)
    wrt.writeheader()
```

```
wrt.writerows(items)
```

(3) 运行代码并确定新文件 `tooldesc3.csv` 已创建。

(4) 在文本编辑器中打开它并检查 `tooldesc3.csv` 确实已经获得了一个标题行。它看起来应该如下所示：

```
ItemID,Name,Description,Owner,Price,Condition,DateRegistered
1,Lawnmower,Small Hover mower,Fred,$150,Excellent,01/05/2012
2,Lawnmower,Ride-on mower,Mike,$370,Fair,04/01/2012
3,Bike,BMX bike,Joe,$200,Good,03/22/2013
4,Drill,Heavy duty hammer,Rob,$100,Good,10/28/2013
5,Scarifier,"Quality, stainless steel",Anne,$200,09/14/2013,
6,Sprinkler,Cheap but effective,Fred,$80,01/06/2014,
```

示例说明

在第一行导入 `csv` 之后，将域名称定义为一个字符串列表。

然后打开原始文件 `tooldesc2.csv`，用 `DictReader` 将它读入到列表 `items` 中。已经使用 `fields` 列表作为 `DictReader` 的 `fieldnames` 参数来初始化它。

下一步是将它写到一个拥有标题行的新文件中。为此，打开了一个新文件 `tooldesc3.csv` 并创建 `DictWriter` 对象，通过 `fieldname` 参数指定了所需要的域顺序(记住，字典保存的域是无序的)。只是简单地传入相同的 `fields` 列表来读取文件，所以这会维持相同的顺序。然后调用 `writer` 对象的 `writeheader()` 方法以及 `writerows()` 方法将整个 `items` 列表一次写出。

你已经看到了如何使用 `csv reader` 和 `writer` 对象在 CSV 文件格式和 Python 列表之间转换，以及如何使用 `DictReader` 和 `DictWriter` 对象对字典做相同的操作。你已经看到了两个修改 CSV 文件格式以方便处理的示例。`csv` 模块包含了其他一些用来处理非基于 Excel 的 CSV 文件的特性。如果需要，可以在文档中阅读相关内容。

2.3.2 使用 Config 文件

`Config` 文件或人们常说的 Windows“INI”文件有一个可读性很高的格式，它在编程中也很易用。但它们在近些年失宠了，因为 Microsoft 现在提倡使用 Windows Registry 而非 Microsoft 应用正在转向基于 XML 的存储。然而，周围还有大量的遗留应用在使用这种格式(在一个相对干净的 Windows 8.1 系统上搜索 `*.ini` 会发现数百个文件，所以它的生命周期还远未结束！)

这种格式非常适合存储多个相似数据的实例，比如网络上每个节点的设置；或多个选项的种类，比如多种屏幕大小或线上与线下的操作参数。`Config` 格式的缺点是它有时太简单以至于一些复杂数据很难适配这种格式。Python 提供了 `configparser` 模块来读写 `Config` 格式数据。



注意: configparser 模块在 Python 版本 3 中名称发生了变化。在 Python 2 中, 它被称为 ConfigParser(大写), 但是在 Python3 中, 它现在全小写: configparser。如果跨 Python 版本工作, 要注意名称的改变。功能仍保持不变)。

Config 文件的基本结构如下:

```
[DEFAULT]
Option1=value1

[SECTION1]
Option2=value2
Option3=value3

[SECTION2]
Option4=value4
etc.
```

DEFAULT 部分值得注意, 因为其中定义的选项应用于所有之后的部分。格式有很大的灵活性, 它支持空格和缩进、嵌套部分, 以及各种各样的其他变体, 包括支持将一个选项中的值插入到另一个选项。configparser 模块不仅可以处理所有这些还可以处理更多。它将数据转换成字典格式, 或者从字典格式转换成数据。这里的字典与之前描述的用于 CSV 文件的字典格式非常相似。

文档通过一个创建文件并读取的示例清晰地展示了基本用法。因为在这里没有必要再重复这个示例, 所以可以在闲暇时浏览它。现在试一试下面的示例。

试一试: 创建并读取 Config 文件

在这个“试一试”中, 你会创建一个 Config 文件, 可以将它应用于之前讨论的工具借用应用。这个 Config 文件描述了标准设置和任何特定用户的重写值。这些设置仅限于借用期间(用天表示)和物品可以被借用的最大值(默认值是 0, 代表没有期限)。

完成下面的步骤来创建文件:

- (1) 创建一个项目目录并切换到该目录下。
- (2) 启动 Python 解释器并输入下面的代码:

```
>>> import configparser as cp
>>> conf = cp.ConfigParser()
>>> conf['DEFAULT'] = {'lending_period' : 0, 'max_value' : 0}
>>> conf['Fred'] = {'max_value' : 200} # Fred's a bit rough with things!
>>> conf['Anne'] = {'lending_period' : 30} # She is a bit forgetful sometimes
>>> with open('toolhire.ini', 'w') as toolhire:
...     conf.write(toolhire)
...
>>>
```


(3) 在文件夹中检查新文件 `toolhire.ini` 是否已创建。

(4) 在你的文本编辑器中打开这个新文件(但保持你的 Python 会话继续运行)并确认它看起来如下所示:

```
[DEFAULT]
lending_period = 0
max_value = 0

[Fred]
max_value = 200

[Anne]
lending_period = 30
```

创建文件之后, 你现在可以读回一些值。

(5) 回到 Python 解释器会话并输入下面的代码:

```
>>> del(conf) # get rid of the old one
>>> conf = cp.ConfigParser() # create a new one
>>> conf.read('toolhire.ini')
['toolhire.ini']
>>> conf.sections()
['Fred', 'Anne']
>>> conf['DEFAULT']['max_value']
'0'
>>> conf['Anne']['max_value']
'0'
>>> conf['Anne']['lending_period']
>>> conf['Fred']['max_value']
'200'
```

(6) 最后, 输入下面的代码, 研究一下非常规的行为:

```
>>> conf['Joe']
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
    File "C:\Python33\lib\configparser.py", line 954, in __getitem__
      raise KeyError(key)
KeyError: 'Joe'
>>> conf.options('Anne')
['lending_period', 'max_value']
>>> conf.options('DEFAULT')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
    File "C:\Python33\lib\configparser.py", line 667, in options
      raise NoSectionError(section)
configparser.NoSectionError: No section: 'DEFAULT'
>>> conf.defaults()
OrderedDict([('lending_period', '0'), ('max_value', '0')])
>>>
```

示例说明

在第一行中导入模块之后，创建了一个 `ConfigParser` 对象，之后将一个名称/值对的字典赋值给 `DEFAULT` 部分。然后规定了 `Fred` 的借用限制(他有一个损坏东西的记录)以及 `Anne` 的 `lending_period` 的限制(因为她可能忘记返回东西，所以她需要一个提示器)。然后用写模式打开了文件 `toolhire.ini`，并用 `ConfigParser` 对象向文件中写入数据。

在检查文件存在并包含正确数据后，你开始从文件中读取数据。

为此，删除了原始的解析器对象并创建了一个新的解析器对象，重新使用了名称 `conf`。

使用 `conf` 读取 `toolhire.ini` 文件，确认了期望的部分是可用的，并且看到 `DEFAULT` 部分没有被包含在列表中。然后从解析器中读取了一些选项值，并看到即使 `DEFAULT` 部分没有包含在列表中也可以读取 `DEFAULT` 值。此外，当没有对用户显式声明特定选项时，`DEFAULT` 值就会被使用(例如，即使 `Anne` 仅指定了 `lending_period`，还是会为她返回默认的 `max_value`)。

然后你进一步了解解析器是如何处理错误情况的。第一次尝试是访问用户并没有提供的值，这会产生典型的字典 `KeyError` 错误。之后你读取 `Anne` 可用的选项，发现解析器返回了默认值以及那些显式定义的值。你也发现 `options()` 对于 `DEFAULT` 部分并不可用，而需要使用显式的 `default()` 方法来获取那些选项。`DEFAULTS` 的非标准行为是使用 `configparser` 时会经历的少数烦恼之一。

2.3.3 操作 XML 和 HTML 文件

作为网页语言，你可能很熟悉 `HTML`。`XML` 作为自描述的数据格式也被广泛使用。`XML` 和 `HTML` 密切相关。`XML` 是一种更加严格定义的格式，而使用计算机来处理它也变得更加容易。`HTML` 则对畸形的内容有很高的容忍度。尽管这让手工或使用特殊编辑器来创建 `HTML` 变得更加轻松，但是想要精确地处理它就要多靠碰运气了。由于 `Web` 浏览器的私有扩展，`HTML` 也有许多变体。所有这些都意味着 `HTML` 解析器的工作非常棘手，而且在面对格式混乱的文件时常常产生不完美的结果。由于 `XML` 用程序的方式更好处理，因此你需要先了解如何解析它，然后再扩展到 `HTML`。



注意：`HTML` 有一种形式叫 `XHTML`，也是有效的 `XML`。随着越来越多的 `Web` 发布工具支持 `XHTML`，它开始出现在网站上。这意味着可以使用 `XML` 解析器去解析 `XHTML` 以及其他形式的 `XML`。然而，`HTML5` 实际上已经弃用 `XHTML`，所以将来它的使用可能会减少而不会增加。

1. 解析 XML 文件

有很多不同的解析器可以被用来解析 `XML`。`Python` 标准库至少包含了 5 个(`dom`、

minidom、expat、ElementTree 和 sax)。这些解析器可以分为两种：一种是读取整个文件到一个名为文档对象模型(Document Object Model, DOM)的树型数据结构中，另一种是读取文件，查找感兴趣(一个事件)的数据项并在数据项被发现时触发一个响应。前者在一组相同数据上做复杂的或多个查询时要更加灵活。后者则使用起来更快也稍微简单些。在本书中，只会介绍两个解析器，它们分别代表了这两种方法。

你将学习的第一个解析器是 sax，它是基于事件解析器的代表。为了理解基于事件解析器的工作原理，请看下面解析纯文本的一些示例：

```
>>> text = """mary had a little lamb
... its fleece was white as snow
... and everywhere that mary went
... the lamb was sure to go"""

>>> def has_mary(aLine):
...     print( "We found: ", aLine)
...
>>> def parse_text(theText, aPattern, function):
...     for line in theText.split('\n'):
...         if aPattern in line:
...             function(line)
...
>>> parse_text(text, 'mary', has_mary)
We found: mary had a little lamb
We found: and everywhere that mary went
>>>
```

在此创建了一些想要解析的文本，然后定义了 has_mary() 函数。你想要在文本中每次发现 mary 时都调用这个函数。

然后创建了基于事件的 parse_text() 解析函数。这个函数会逐行迭代输入文本。当搜索到目标字符串时，在本例中是 mary，它会调用被传入的函数。

当使用 text 字符串和 has_mary() 函数作为参数执行 parse_text() 时，它会打印包含 mary 的两行。

sax 模块和 parse_text() 函数的工作方式非常相似。然而，它使用事件，比如探测一个 XML 元素的开始而不是纯文本模式。它接受一个 XML 源文本、一批事件和相关联的事件处理程序函数。然后逐部分地处理 XML 文本，而当它找到一个给定事件的匹配时，就会调用关联的处理程序来处理。解析器并不存储 XML 数据，它只是简单的迭代。如果需要回访之前的数据，需要再次解析整个文件。

为了研究 sax 解析器，需要一个 XML 文件。可以在 Chapter2.zip 文件的 ToolhireData 文件夹中找到 toolhire.xml。这只是之前你使用 toolhire.xlsx 电子表格导出的一个文件。下面展示了该文件的一小部分，包括你将会提取的部分。为了可读性，在此做了一些编辑：

```
<?xml version="1.0"?>
<?mso-application progid="FileName_Excel.Sheet"?>
```



```

<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
...
<Worksheet ss:Name="Sheet1">
  <Table ss:ExpandedColumnCount="1025" ss:ExpandedRowCount="7"
x:FullColumns="1"
  x:FullRows="1" ss:DefaultRowHeight="15">
    <Column ss:AutoFitWidth="0" ss:Width="36"/>
    ...
    <Row ss:StyleID="s36">
      <Cell><Data ss:Type="String">ItemID</Data></Cell>
      <Cell><Data ss:Type="String">Name</Data></Cell>
      <Cell><Data ss:Type="String">Description</Data></Cell>
      <Cell><Data ss:Type="String">Owner</Data></Cell>
      <Cell><Data ss:Type="String">Borrower</Data></Cell>
      <Cell><Data ss:Type="String">DateLent</Data></Cell>
      <Cell><Data ss:Type="String">DateReturned</Data></Cell>
    </Row>
    <Row>
      <Cell><Data ss:Type="Number">1</Data></Cell>
      <Cell><Data ss:Type="String">LawnMower</Data></Cell>
      <Cell><Data ss:Type="String">Small Hover mower</Data></Cell>
      <Cell><Data ss:Type="String">Fred</Data></Cell>
      <Cell><Data ss:Type="String">Joe</Data></Cell>
      <Cell ss:StyleID="s37"><Data ss:Type="DateTime">
2012-04-01T00:00:00.000</Data></Cell>
      <Cell ss:StyleID="s37"><Data ss:Type="DateTime">
2012-04-26T00:00:00.000</Data></Cell>
    </Row>
    ...
  </Worksheet>
</Workbook>

```

假设想要找出借出的平均期限。使用 `sax` 只提取每个项的 `DateLent` 和 `DateReturned` 域值并把它们作为一个元组存储在日期列表中。可以在之后处理这些日期，找到每个借出物品的持续时间。

为了初始化解析器，需要创建处理程序并指定感兴趣的事件。`sax` 实际上用处理程序对象来结合事件和函数，这个处理程序对象是 `xml.sax.handler.ContentHandler` 类的一个实例，更确切地说是它的一个子类。已经存在几个预定义的处理程序子类，其中一个用来处理错误。这种方法的优点是许多默认方法都已经被定义了而其他的都可以被轻松重写，例如，在解析最开始时调用的 `startDocument()` 对于设置状态变量非常有用，等等诸如此类。对于简单的 XML 解析任务，通常要创建一个 `ContentHandler` 的自定义子类，然后编写自己的 `startElement()`、`endElement()` 和可能的 `character()` 方法。

通过检查 XML 文件，可以看到需要的数据被包含在一个 `<Data>` 元素中，并且通过设置为 `DateTime` 的 `ss:Type` 特性确定。实际的数据是 `<Data>` 开始和结束标签之间的字符数据。所以，期待的事件顺序是 `startElement()`，然后是 `character()`，最后是 `endElement()`。

ToolHireHandler 类的代码看起来如下所示(如 Chapter2.zip 的 ToolHire 文件夹中的 toolhiresax.py 文件):

```
import xml.sax
import xml.sax.handler

class ToolHireHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
        super().__init__()
        self.dates = []
        self.dateLent = ''
        self.dateCounter = 0
        self.isDate = False

    def startElement(self, name, attributes):
        if name == "Data":
            data = attributes.get('ss:Type', None)
            if data == 'DateTime':
                self.isDate = True
                self.dateCounter += 1
            else:
                self.dateCounter = 0

    def endElement(self, name):
        self.isDate = False

    def characters(self, data):
        if self.isDate:
            if self.dateCounter == 1:
                self.dateLent = data
            else:
                self.dates.append((self.dateLent, data))

if __name__ == '__main__':
    handler = ToolHireHandler()
    parser = xml.sax.make_parser()
    parser.setContentHandler(handler)
    parser.parse('toolhire.xml')
    print(handler.dates)
```

初始化程序调用了父类的初始化程序, 然后设置了在解析中和各种方法中需要使用的各种数据特性。它还创建了一个空的 dates 列表来存放结果。

主要的解析方法是 startElement() 方法。它会寻找 Data 元素, 一旦发现, 它会通过只选择那些 ss:Type 特性为 DateTime 的元素来优化搜索(需要手动查找 XML 文件来确定这些值)。因为在单行中可以最多有两个日期, 所以可以使用 self.dateCounter 来追踪正在处理此行中的哪个日期。可以使用 self.isDate 值告诉 character() 方法它在一个日期元素中。如果数据不是 DateTime 类型, 则将 self.dateCounter 重置为 0。

`endElement()` 方法保证 `self.isDate` 标签被重置为 `False`，为下一个将要到来的 `startElement()` 事件做准备。

无论何时碰到标签元素外面的内容，`character()` 方法都会被调用。你只是对日期信息感兴趣。所以如果 `self.isDate` 标签没有被设置，则只需要忽略字符数据。如果数据是一个日期，则会检查它是不是第一个日期。如果是，则将它存储在 `self.dateLent` 特性中；如果它是第二个日期，则将两个日期都存储在 `self.dates` 列表。如果只发现了一个日期，则不会第二次调用字符处理程序。而日期也不会被添加到日期列表中。这样可以保证只存储计算持续时间所需要的成对日期。

最后，底部的驱动代码创建了处理程序和解析器实例。然后将解析器中的处理程序设置为 `ToolHireHandler` 实例并对 XML 文件执行 `parse()` 操作。在解析完成之后，它打印出从处理程序获得的日期。

你会在本节最后的“试一试”中再次练习使用 `ElementTree` 基于 DOM 的解析器。其中会对比这两个技术。但是首先会解析 HTML，因为标准库 HTML 解析器在风格上与 `sax` XML 解析器非常相似。

2. 解析 HTML 文件

标准库提供了 `html.parser` 模块来解析 HTML。它的工作方式与 `sax` 解析器类似，也是事件驱动的。它用起来要稍微容易些，因为它内部只含有一个定义处理程序方法的类。为了展示它的工作原理，你还要从 `toolhire.xlsx` 电子表格中提取出日期，但是这一次导出为 HTML 格式。可以在 `ToolhireData/toolhire_files` 文件夹下的 `zip` 文件中找到这个文件，文件名为 `sheet001.htm`。

该文件某种程度上看起来如下：

```
<html xmlns:v="urn:schemas-microsoft-com:vml"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns:x="urn:schemas-microsoft-com:office:excel"
xmlns="http://www.w3.org/TR/REC-html40">

<head>
<meta http-equiv=Content-Type content="text/html; charset=windows-1252">
<meta name=ProgId content=Excel.Sheet>
...
<body link=blue vlink=purple>

<table border=0 cellpadding=0 cellspacing=0 width=752 style='border-collapse:
collapse;table-layout:fixed;width:564pt'>
<col width=64 style='width:48pt'>
<col width=115
style='mso-width-source:userset;mso-width-alt:4205;width:86pt'>
...
<tr class=x166 height=21 style='height:15.75pt'>
<td height=21 class=x166 width=64
```



```

style='height:15.75pt;width:48pt'>ItemID</td>
    <td class=xl66 width=115 style='width:86pt'>Name</td>
    <td class=xl66 width=153 style='width:115pt'>Description</td>
    <td class=xl66 width=80 style='width:60pt'>Owner</td>
    <td class=xl66 width=120 style='width:90pt'>Borrower</td>
    <td class=xl66 width=99 style='width:74pt'>DateLent</td>
    <td class=xl66 width=121 style='width:91pt'>DateReturned</td>
</tr>
<tr height=20 style='height:15.0pt'>
    <td height=20 align=right style='height:15.0pt'>1</td>
    <td>LawnMower</td>
    <td>Small Hover mower</td>
    <td>Fred</td>
    <td>Joe</td>
    <td class=xl65 align=right>4/1/2012</td>
    <td class=xl65 align=right>4/26/2012</td>
</tr>
...
</tabular>
</body>
</html>

```

可以看到，日期有一个名为 xl65 的特殊类。这意味着可以像之前的 XML 示例那样，使用类似的方法来查找拥有这个类属性的<td>标签。

HTMLParser 类的工作方式与 saxContentHandler 类非常相似。它含有对应 HTML 文档元素的方法。在这个示例中，重写了 handle_starttag()、handle_endtag()和 handle_data()方法。这些方法直接类似于 XML 的 startElement、endElement 和 character 方法。

可以在 zip 文件 ToolHire 文件夹下的 toolhirehtml.py 文件中发现这个示例的代码。代码如下所示：

```

import html.parser

class ToolHireParser(html.parser.HTMLParser):
    def __init__(self):
        super().__init__()
        self.dates = []
        self.dateLent = ''
        self.isDate = False
        self.dateCounter = 0

    def handle_starttag(self, name, attributes):
        if name == 'td':
            for key, value in attributes:
                if key == 'class' and value == 'xl65':
                    self.isDate = True
                    self.dateCounter += 1
                    break
            else:
                self.dateCounter = 0

```

```

def handle_endtag(self, name):
    self.isDate = False

def handle_data(self, data):
    if self.isDate:
        if self.dateCounter == 1:
            self.dateLent = data
        else:
            self.dates.append((self.dateLent, data))

if __name__ == '__main__':
    htm = open('sheet001.htm').read()
    parser = ToolHireParser()
    parser.feed(htm)
    print(parser.dates)

```

如果将该示例与 sax 示例进行对比,就会发现方法中的代码几乎是相同的。HTMLParser 把它的特性展示为一个元组列表。迭代这个列表,通过查找类特性值为 xl65 来确定一个日期域(注意是 xl(字母 L)而不是 x1(数字 1),记住这是从 Microsoft Excel 导出的,类名也是如此)。解析器很贴心地考虑了混合大小写 HTML 标签或将标签名转换为小写,所以你不需担心这一点。同时,它也尽量让糟糕格式的 HTML 变得有意义,尽管它不是很完善而且非常糟糕的代码可以让它犯错。

在本节的结尾,将通过读取数据来了解另一个 Python XML 解析器。

该解析器是 ElementTree, 在下面的“试一试”中将研究它。

试一试: 使用 ElementTree 解析 XML(toohireET1.py 和 toohireET.py)

在这个“试一试”中,使用 ElementTree XML 解析器提取之前示例中一样的日期。你也要为电子表格中的物品计算平均借出时间。为此,完成下面的步骤:

- (1) 创建一个新项目文件夹并将之前使用过的 toolhire.xml 文件复制到这个文件夹中。
- (2) 打开 IDE 或文本编辑器,输入下面的代码并将它保存为 toolhireET.py(或者从 zip 文件中提取出 toolhireET1.py):

```

import xml.etree.ElementTree as ET
import datetime as dt

```

```

def parseDates(filename):
    dates = []
    rows = []
    dom = ET.parse(filename)
    root = dom.getroot()
    for node in dom.iter('*'):
        if 'Row' in node.tag:
            rows.append(node)
    for row in rows:
        row_dates = []

```

```

    for node in row.iter('*'):
        for key,value in node.attrib.items():
            if 'Type' in key and 'DateTime' in value:
                row_dates.append(node.text)
    if len(row_dates) == 2:
        dates += row_dates
    return dates

def main():
    print(parseDates('toolhire.xml'))

if __name__ == '__main__':
    main()

```

(3) 打开操作系统控制台窗口，并切换到你的项目文件夹。

(4) 使用 python3 toolhireET.py(或者 toolhireET1.py，如果使用 zip 文件)在控制台运行文件。

(5) 检查输出是否如下：

```

['2012-04-01T00:00:00.000', '2012-04-26T00:00:00.000',
'2012-09-05T00:00:00.000',
'2013-01-05T00:00:00.000', '2013-07-03T00:00:00.000',
'2013-07-22T00:00:00.000',
'2013-11-19T00:00:00.000', '2013-11-29T00:00:00.000',
'2013-12-05T00:00:00.000']

```

(6) 返回到 IDE 或编辑器，添加 calculateAverage() 函数并如下所示修改 main()(或者从 zip 文件中加载 toolhireET.py)：

```

def calculateAverage(dates):
    loan_periods = []
    while dates:
        lent = dates.pop(0).split('T')[0]
        ret = dates.pop(0).split('T')[0]
        lent_date = dt.datetime.strptime(lent,'%Y-%m-%d')
        ret_date = dt.datetime.strptime(ret,'%Y-%m-%d')
        loan_periods.append( (ret_date - lent_date).days )
    average = sum(loan_periods)/len(loan_periods)
    return average

def main():
    dates = parseDates('toolhire.xml')
    avg = calculateAverage(dates)
    print('Average loan period is: {} days'.format(avg))

```

(7) 保存文件并运行代码。你应该看到一条消息，告诉你平均的借出时间是 44 天。

示例说明

首先，导入了 `ElementTree` 解析器并将它的别名设置为 `ET`。然后为了之后的日期计算，导入了别名为 `dt` 的 `datetime` 模块。

之后创建了 `parseDates()` 函数。该函数首先将 XML 文件解析到一个 DOM 中。注意，`parse()` 函数接受一个文件名作为参数，而不是一个文件对象。然后使用 `getroot()` 方法从 DOM 中获得根节点。使用了 `iter()` 方法来查找所有的文件，这是通过 * 参数指定的。接下来检查了节点，如果在它的标签中存在 `Row`，就将这个节点添加到 `rows` 列表中。

在构建了 `rows` 列表之后，你深入到每一行中检查每个节点，检查它们的特性，寻找 `key` 为 `Type` 和 `value` 为 `Datetime` 的节点。一旦发现，就将日期节点插入到 `row_dates` 列表中。在每一行的结束，如果 `row_dates` 列表包含两个日期，就将它添加到 `dates` 列表中。否则，就忽略它。在函数的最后，返回了最终的 `dates` 列表。

然后，测试了函数并检查输出是否是期待的日期列表。

接下来，添加了新函数 `calculateAverage()` 并相应地修改了 `main()` 函数。

在 `calculateAverage()` 函数中，初始化了一个列表来保存每一次借用的时间。然后迭代日期列表并把它们成对地提取出来。你知道它们都是成对的，因为你在 `parseDates()` 函数中只添加成对的日期。提取过程涉及对字母 `T` 进行日期字符串的分割并只保存字符串的第一部分(需要分割字符串，因为 `datetime.strptime()` 方法不能处理小数的秒值)。下一步是使用 `strptime()` 方法将日期字符串转换成 `datetime` 对象。然后使用 `datetime` 对象的算术能力计算 `timedelta` 来表示借用期限，并将 `days` 值存储在 `loan_period` 列表中。最后，计算存储的时间段的平均值并返回结果。

一些应用不适合生成数据文件。在这些情况下，可能需要通过应用编程接口(API)与程序交互。下一节将介绍如何实现这一点。

2.4 使用 ctypes 和 pywin32 访问原生 API

一些应用或操作系统函数不容易通过常规 Python 代码访问，因为没有可以从 Python 中调用的 Python API 或暴露出来的用户友好的操作。`ctypes` 模块可以提供一个可选的访问方法。这种方法向 Python 提供了用于构建应用的 C 代码库。在 Windows 中，这些库通常是一组 DLL 文件。而在 UNIX 中，它们是一组共享对象库。`ctypes` 让你将这些库加载到你的应用中，并直接从 Python 中调用它们的函数。当然，这仅在你知道库中有哪些函数、需要什么参数以及返回值时才有效。这可能被发布，因为需要反复试验或者逆向工程。反过来，逆向工程可能被生产者或供应商所禁止。然而，如果库有一个已发布的接口，`ctypes` 就提供了一种虽不常规但有效的访问方法。



注意：使用 `ctypes` 需要一些 C 语言编程的基础知识。如果没有这项技术，你可能想跳过这部分，因为它可能没有什么意义。另一方面，浏览一下它能让你知道它有哪些用途，有些你可能会需要。

当使用 `ctypes` 时，你就把 Python 解释器的安全网抛在一边了。记住，你在操作原始操作系统库时，有时会直接访问内存地址。这些库也直接操作原始文件系统和输入/输出流，所以它们可能不会展示你使用 IDE，比如 IDLE 或 Pythonwin 时所期待的结果。

如果犯了错误，则可能会轻易导致 Python 解释器崩溃。在极端情况下，甚至可能导致操作系统崩溃。这就是为什么在万不得已的情况下你才使用 `ctypes` 和类似东西的原因。只有当其他方法都失败时，才去使用它。

另一个包是 `pywin32`，它在 ActiveState 发行版本的 Python 中被默认安装。在其他发行版本中也可以下载安装。这个包提供了对 Windows 原生库的访问，尤其是对任何 Microsoft 组件对象模型(COM)接口的访问。特定于 Windows，`pywin32` 要比 `ctypes` 更易用。一般来说，`ctypes` 可以在任何操作系统上工作。对于使用 `ctypes` 的警告同样可以应用在 `pywin32` 上。

2.4.1 访问操作系统库

通常，一个文档编写良好的领域就是操作系统应用编程接口(API)。它被暴露在标准系统库中。在本节中，你会使用操作系统库来执行一些非常简单的任务，但是这些任务通过 Python 的 `os` 模块是无法完成的。这个模块对于 Windows 用户尤其有用，因为 Windows 下许多类 UNIX 的特性在 `os` 模块中是不可用或部分可用的。直接通过 `ctypes`(或 `pywin32`)来访问 Win32 API 通常是唯一的选择。



注意：`ctypes` 既不能使用静态库也不能使用 C++库，除非函数已经被显式地从 C++代码导出为 C 函数。

下面各小节展示了 `ctypes` 在 Windows 和 Linux 系统上的使用，除了将初始引用指向 C 库，其他原理是相同的。

1. 在 Windows 上使用 `ctypes`

在 Windows 系统上，基本 C 库存在于 `msvcrt` 库中。`msvcrt.dll` 中的一些函数被包含在 Python `msvcrt` 模块中，这些函数主要是关于控制台输入/输出操作的。但是更多的函数并没有被包含在其中。可以使用下面的代码通过 `ctypes` 轻松地访问原生 `msvcrt` 库：

```
>>> import ctypes as ct
>>> libc = ct.cdll.msvcrt # Windows only
```

一旦拥有一个标准库的引用，就可以调用熟悉的 C 函数。唯一复杂的地方是需要确认参数与 C 是类型兼容的。通常来说，整型参数用起来方便些，但是字符串通常需要被显式地标记为字节字符串，而浮点数需要一个特殊的 ctypes 类型转换。许多类型转换函数被包含在 ctypes 中。可以在模块文档中找到一个完整的列表。以下是两个示例：

```
>>> libc.printf(b"%d %s %s hanging on a wall\n", 6, b"green", b"bottles")
6 green bottles hanging on a wall
34
>>> libc.printf(b"Pi is: %f\n", ct.c_double(3.14159))
Pi is: 3.141590
16
```

注意，使用 b 来指示一个字节字符串，而在第二个示例中使用的是 ctypes.c_double() 转换函数。同时，注意 sprintf() 的返回值是打印的字符的数量。它在信息被打印之后被显示出来。

许多 C 函数需要指向数据的指针(实际上是内存地址)作为参数。ctypes 使用 byref() 函数帮助可以实现这一点。可以创建一个给定类型的对象，然后使用 byref() 将这个对象传入你想要执行的 ctypes 函数。下面这个示例使用 sscanf() 从字符串读取整型值到 Python 变量中：

```
>>> d = ct.c_int()
>>> print(d.value)
0
>>> libc sscanf(b"6", b"%d", ct.byref(d))
1
>>> print(d.value)
6
```

接下来，你会看到 Windows 库中一个更加实用的函数：msvcrt._getdrives()。这个函数返回 Windows 系统上一个可用的驱动器列表，这一点使用 Python 标准库是不容易实现的。唯一复杂的地方是返回的列表是一个位掩码。所以需要编写一个循环来找出哪些位已被设置并将位的位置映射到一个盘符上。代码如下：

```
>>> drives = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> drivelist = libc._getdrives()
>>> for n in range(26):
...     mask = 1 << n # use left bit shifting to build a mask
...     if drivelist & mask: print (drives[n], 'is available')
...
C is available
D is available
E is available
P is available
```


微软开发者网站(msdn.microsoft.com)上有关于标准 Windows 库函数的完整文档。

2. 在 Linux 上使用 ctypes

也可以在非 Windows 系统上使用 ctypes。下面是一个在 Linux 系统上使用 printf() 的示例。它是通过标准 C 库 libc.so.6 访问的(如果能找到实现了标准 C 库函数的库名, 也可以使用其他类 UNIX 操作系统)。

```
>>> import ctypes as ct
>>> libc = ct.CDLL('libc.so.6')
>>> libc.printf(b"My name is %s\n", b"Fred")
My name is Fred
16
```

如果使用 Linux libc, 之前部分的 printf() 和 sscanf() 示例也能够正常运行, 另外, byref() 函数和各式各样的类型转换函数也能够正常运行。

2.4.2 使用 COM 访问 Windows 应用

如果能得到库内容文档, 那么访问应用库和访问操作系统系统库也很容易。然而, 文档并不是永远可用的。在 Windows 上的另一个选项就是使用操作系统函数访问 COM 对象, 然后从 Python 操纵 COM 对象。遗憾的是, COM 是一个复杂的技术, 而且经过一段时间后扩展为包括一些特性, 比如在网络上分布以及各式各样的数据访问机制。COM 对象的文档通常很稀少而且很难找到, 这更增加了难度。尽管如此, COM 通常是自动化 Windows 应用的最有效的选项。

在 Python 中使用 COM 对象的最简单方式是使用由 Mark Hammond 编写的 pywin32 包。它可以在 SourceForge 网站上下载到, 或者作为标准被包含在 Python 的 ActiveState 发行包中。接下来的“试一试”中演示了使用 pywin32 打开预先加载了本章前面使用过的 toolhire.xlsx 文件的 Excel。

试一试: 使用 COM 呈现一个 File-Open 对话(toolhireCOM.py)

这个“试一试”演示了在 Python 中如何使用 Excel COM 接口打开一个应用并让用户选择文件。如果在使用 Windows, 请依照如下这些步骤(这个示例仅在 Windows 下有用):

(1) 如果没有安装 ActiveState 版本的 Python, 请从 SourceForge 网站 <http://sourceforge.net/projects/pywin32/> 上下载并安装 pywin32 扩展包。

(2) 打开 Python IDE, 创建一个名为 toolhireCOM.py 的新文件(或者从 zip 文件中加载它)。输入下面的代码(确保将文件路径设置为你自己的文件位置):

```
import win32com.client as com
# set the file path as required on your PC
filepath = r"D:\PythonCode\Chapter2\CSVexamples\toolhire.xlsx"
fileopen = 1 # found by trial and error!
app = com.Dispatch("Excel.Application")
app.Visible = True
```

```
fd = app.FileDialog(fileopen)
fd.InitialFileName = filepath
fd.Title = "Open the toolhire spreadsheet"
if fd.Show() == -1:
    fd.Execute()
```

- (3) 保存并运行该文件。
- (4) 在打开的对话框中单击 OK 按钮。
- (5) 确认电子表格包含了之前使用过的电子表格的数据。

示例说明

在导入 win32com.client 模块并赋予它别名 com 后，将文件路径设置到一个变量中。如果需要，很容易更改(注意：你应该使用自己文件夹的路径，而不要使用此处的路径)。下一行设置 filemode 变量为 1。这决定会打开什么样的对话。在这个示例中，打开了一个 File-Open 对话(这个值是通过反复试验找到的，有效值在 1 到 4 之间)

然后使用 Dispatch() 函数创建了 Application COM 对象并通过设置它的 Visible 属性为 True 让窗口可见。这时候，窗口出现在屏幕上，但是没有往常的网格。这是因为 Excel 实际上将网格存储在另一个名为 Workbook 的 COM 对象中。如果知道哪个文件是你感兴趣的文件，就可以创建一个 Workbook 对象(或者更确切地说是一组 Workbooks 或制表符)并打开文件而不是使用一个对话框。Workbook 对象包含 Cells。如果想要创建或修改电子表格中的数据，则应该使用 Cells。

下一步，使用 Application 的 FileDialog() 方法创建了一个对话对象，它以 filemode 值作为参数。然后你为该对象设置了两个属性，确保它在正确的地方被打开并且拥有一个有意义的名称。

最后，调用了对话的 Show() 方法将对话框显示在屏幕上。用户可以使用所有常用的功能。如果用户选择 OK 按钮，返回值是 -1。在这种情况下，可以调用对话对象的 Execute() 方法来打开(或保存，如果必要的话)被选中的文件。此时电子表格中填充的就是 Workbooks 的内容，并且跟你平常看到应用时的样子一样。

至此，你已经学习了很多在脚本程序中整合不同应用的技术。下一节将介绍一些关于如何将它们集合起来完成一个脚本项目的建议。

2.5 涉及多应用的自动化任务

在本章的开始，脚本被定义为通过协调其他程序或应用的行为来实现某一任务的可执行文件。目前，你已经认识了几个支持模块。这些模块有助于你与外部程序进行交互。但我们还没有讨论如何自动化一个完整的工作流程。

在处理一个工作流程自动化的项目时，你通常会观察人工操作的流程。通过观察，你能了解整个流程中使用的系统、发生的动作，以及输入和输出的数据。然后，你会使用任

何可用的自动化功能去复制每一个系统和进程。此外，你应该去掉那些单纯用于方便用户的步骤。例如，很多系统会把数据格式化以方便阅读，但这些数据只是中间结果。如果计算机可以读取未格式化的数据，格式化的步骤就是不必要的。一旦确定了必要步骤和需要的系统和工具，就可以考虑自动化功能了。

本章讨论了一些开发多应用脚本的指南。这些指南可以使开发代价减到最小。通常，请按照下面讨论的顺序使用技术。

2.5.1 使用 Python

Python 有很多支持模块。这些模块允许你在代码中直接实现操作系统的功能和命令。还有一些模块可以帮助操纵不同类型的文件和网络协议。例如，Python 中有的模块可以直接操纵 Windows 注册表和 UNIX 密码文件。使用这些模块可以避免调用其他外部程序。这种 Python 解决方案高效灵活，且易于维护。所以，在可能的情况下，Python 应该永远是第一选择。

2.5.2 使用操作系统工具

操作系统提供了许多工具和命令来管理。许多管理工具拥有命令行接口(CLI)。通过 subprocess 模块，Python 可以轻松地在代码中调用这些接口。虽然没有交互过程的工具是最方便的，但这意味着需要使用数据文件作为中间步骤。当程序失败时，数据文件可以作为恢复节点。只需要在最后一个成功的步骤重启程序就可以了。

2.5.3 使用数据文件

许多工具和操作系统命令使用 Config 文件来控制它们的功能。在运行命令前，可以通过创建或修改这些 Config 文件来控制程序的行为。这可以避免与程序实时交互的复杂性。此外，这些工具可以处理输入文件并产生输出文件。这样就不需要在交互中提供数据。之前你已经了解了 Python 可以解析多种数据格式，所以 Python 代码可以创建或读取这些文件。

2.5.4 使用第三方模块

许多流行应用都有第三方模块。这些模块可以协助与应用的交互，或者直接操纵应用的数据文件。微软 Excel 就是一个很好的示例。它有数个模块可以帮助操作电子表格。对于一些专有文件格式，可以使用第三方模块来操纵。用你最喜欢用的搜索引擎可以找到这些模块。在搜索的关键字中加入应用的名称，比如 Python 和模块，这样会帮助你迅速获得你想要的。

对于这个方法，有一点需要注意。第三方模块经常基于旧版本 Python 工作，但没有升级到 Python 的最新版本。大部分这类模块是开源的，并且拥有宽松的许可条件。所以，可以自己升级第三方模块的代码。如果工程太大，也可以只是把需要的代码复制到你的项目。

当然，代码的贡献者中应当包含原作者。

2.5.5 通过命令行接口与子进程交互

当一个工具有命令行接口但是不能通过数据文件驱动，则还可以通过 `subprocess` 模块以及 `stdin` 和 `stdout` 与程序进行交互。这点已经在本章的子进程管理部分有清晰的阐述。然而，这种策略可能会非常复杂，因为你必须预先知道程序产生的所有可能的响应或输入请求。同时，错误处理也会很难控制。如果一个程序产生了预期之外的交互，则可能只有终止脚本并手动恢复，别无他法。这就是为什么在完全可能的情况下，数据文件是更好的选择。



注意：在 Mac OS X 系统上，有另外一种技术。本书并没有详细介绍它。但是它对于脚本化 Mac 应用是非常有用的。它基于 AppleScript 技术和它的命令行接口 `osascript`。通过编写小的 AppleScript 程序并通过 `osascript` 从 Python 中调用它们，打个比喻，你常常可以让 Apple 程序一起跳舞。有一些第三方模块可以用来与 `osascript` 交互，但是也可以使用 `subprocess` 模块直接运行它。

有一个第三方模块叫 `pexpect`。当与一个基于控制台的外部程序进行交互时，这个模块可以让它变得轻松。这个模块会从目标程序中搜寻期望的提示字符串，然后允许程序员发送指令响应。这在登录会话和类似的交互过程中非常有用。

2.5.6 为基于服务器的应用使用 Web 服务

一些应用把网络服务作为一个接口选项。尽管需要权衡往往更慢的性能和解析这些服务使用的 XML 或 JSON 数据格式所带来的复杂性，但这通常是使用第三方模块的一个很有吸引力的替代。Web 服务会在第 5 章有更加详细的讨论。

2.5.7 使用一个原生代码 API

如果需要控制的应用提供了一个 C 库作为 API，那么可以使用 `ctypes` 从 Python 访问它。使用这个方法时，可能遇到的最大问题就是寻找这个 API 的不错的文档。如果文档存在，这可以是一个非常有效的技术。但是，如果不存在，则可能意味着痛苦的反复尝试。在这些情况下，Python 交互式提示符是一个非常重要的工具。

对于 Windows 应用，通常可以找到一个 COM 接口并使用 `win32` 包来访问它。对于使用 `ctypes`，缺少文档是最大的障碍。

2.5.8 使用 GUI 机器人学

对于没有 API 的 GUI 应用，最后一个选择就是通过给应用发送用户事件消息与 GUI

本身进行交互。这种事件包括按键、鼠标单击等。这种技术被称为机器人学(robotics)，因为你在 Python 程序中模仿一个人类用户。实际上，这是在之前一节讨论的源代码的一种扩展。但它是在一个更低的级别操作。

这是一个令人沮丧的技术。它是非常易于出错的，并且非常容易被正在控制的应用中的改变所影响。比如，如果一次升级改变了屏幕的布局，你的代码很可能会中断。由于编写代码非常困难而解决方案也非常脆弱，因此应该避免使用这个方法，除非其他方法都失败了。

2.6 本章小结

本章介绍了如何去自动化不同应用或操作系统工具的任务。你学习了 Python 标准库含有几个强大的模块可以帮助实现这一切。例如，os、os.path、shutil 和 glob 模块可以提供关于计算机资源的大量信息，并且帮助你直接在 Python 中管理文件。

subprocess 模块提供了一种在脚本中启动命令程序并与之交互的机制。

time、datetime 和 calendar 模块可以辅助时间相关的任务和计算。

time.sleep()函数可以在等待其他进程完成时暂停脚本的执行。

你也学习了可以被生成或用作应用输入的常用数据文件。通过使用模块，比如 csv、configparser、htmllib 和 xml.etree，它们也可以被 Python 创建或读取。

如果没有其他可用的访问方式，也可以使用 ctypes 来访问动态库提供的 C 函数。在 Windows 上，也有可用的类似的以 COM 接口方式提供的函数。pywin32 模块在某种程度上简化了访问。通常，这些技术比使用数据文件或调用 subprocess 函数更加复杂。

最后，你回顾了可用选项的优点和缺点，包括对于 GUIs 的最后的选项，就是发送操作系统事件到应用窗口。这个选项非常难以编写代码，所以应该在所有其他方式都已经试过并失败时才可以使用它。

练习

1. 使用 os 模块，对于你的计算机看看能发现什么信息。一定要阅读 Python 文档中 os 和 stat 模块的相关部分。

2. 试着将一个名为 find_dirs()的新函数添加到 file_tree 模块中，这个函数搜索能够匹配给定正则表达式的目录。结合这两个函数创建第三个函数 find_all()，它既会搜索文件也会搜索目录。

3. 创建另一个函数 apply_to_files()，该函数将一个函数参数应用到所有匹配输入模式的文件上。例如，可以使用这个函数删除所有匹配某个模式(比如*.tmp)的文件。如下所示：

```
findfiles.apply_to_files('.*\\.tmp', os.remove, 'TreeRoot')
```

4. 编写一个程序，迭代前 128 个字符并显示一个消息，表示值是否是一个控制字符(其序数值在 0x00 和 0x1F 之间以及 0x7F 的字符)。使用 ctypes 访问标准 C 库并调用 iscntrl()

函数来确定给定字符是否是一个控制字符。注意，该函数并非 Python 中字符串类型的内置测试函数。

本章所学知识

主 题	关 键 概 念
脚本	一个任务的自动化涉及许多工具和应用。Python 被用作胶水将这些工具绑在一起，将数据格式转换成兼容形式，同步活动，并且如果必要的话可以作为一个伪用户驱动功能
操作系统环境	当操作系统运行进程时，它创建了一个环境，其中包括某些配置细节，如进程优先级、它的主目录、文件权限和格式。脚本通常需要在启动程序之前自定义环境，以确保它以正确的方式执行
进程和子进程	操作系统运行的程序被称为进程。一个应用可能包含一个进程层次结构。在这个层次结构中，一个顶层进程孵化出多个子进程。默认情况下，子进程继承其父进程的环境。脚本经常把其他程序作为子进程启动
树的遍历	文件系统以一个树结构存在，包括一个根节点和附着在根节点上的子树。可以递归地向下遍历这个结构直到叶节点。叶节点就是独立的文件。脚本经常需要在文件系统上的给定子树上处理多个文件
绝对日期和时间	历史上一个固定的日期和时间。一个日期，比如 1776 年 7 月 4 日，就是一个绝对日期
相对日期和时间	相对于另一个日期或时间的日期或时间。通常被表示为一个时间段，比如 3 小时，或一个重复的日期或时间，比如每天的第三个小时或每月的第一天
解析器	一个用来将结构化的数据解析成其组成部分的函数。解析器可以基于几种不同的算法，而最常用的类型就是基于事件和基于树的类型。Python 支持这两种风格的 XML 解析
库	编程语言将可复用的代码放在代码库中。这样就让它们可以被他人所用。这些在概念上类似于 Python 模块。但是在编译语言中，这些是使用特殊工具生成的，而且可以被静态地或动态地链接到一个应用中。ctypes 可以访问动态链接的 C 库
COM	Windows Common Object Model(COM)机制允许外部应用(或经常是内部宏语言)操纵程序的功能。pywin32 包简化了 Python 对 COM 对象的访问

第 3 章

管理数据

本章主要内容:

- 什么是数据持久化
- 如何在文件中存储数据
- 如何在数据库中存储数据
- 如何用数据库搜索、排序和访问数据
- 数据存储的其他选项

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/python-projects 的 Download Code 选项卡下找到。第 3 章代码位于 Chapter 3 download, 名为 Chapter3.zip, 每个文件都是根据本章提到的代码文件名命名的。

在很多场景中, 需要在程序运行期间存储数据。你想要存储的数据可以是本地状态信息, 比如电子书阅读器的当前位置或当前的工作文件名。也可以是管理数据, 比如用户名和密码, 或者服务器地址。有时, 它可能是大量面向业务的数据, 比如消费者订单、库存或地址信息。许多业务应用仅仅包含创建、查看和编辑所存储数据的机制。

以这种能够让程序的后续调用使用的方式存储数据的能力被称为数据持久化(data persistence), 因为数据持续的时间要长于创建它的进程的生命期。为了实现数据持久化, 需要将数据存储在某些地方, 或者在文件中或者在数据库中。

本章有点像在介绍计算存储技术的历史。这是因为随着时光流逝, 需要存储的数据已经变得而且继续变得更加复杂。你现在有范围广泛的可用技术。这些技术覆盖了每一个存储需求, 从一个简单的配置设置到表示数千逻辑实体的复杂分布式数据源。在本章, 你会了解到存储数据不同的可用选择以及每一个选择的优点和缺点。在这个过程中, 会看到 Python 模块如何有助于完成这种基本的编程任务。

3.1 使用 Python 存储数据

最简单的存储就是纯文本文件。在第 2 章，你已经看到如何使用各式各样的文本文件存储数据，比如 CSV 和 XML，以及如何存储未格式化的文本。如果需要仅在程序关闭时存储数据并在程序再次启动时读回它，这些格式很好。这种情况让这些格式非常适合配置数据或应用状态信息。但是当需要处理大量非序列化数据或搜索特定记录或值域时，这些平面文件格式就不起作用了。对于这些情况，需要一个数据库。

数据库只是一个数据存储系统。它允许你创建、读取、更新和删除单独的记录。这四个基本数据管理函数通常称为一个 CRUD 接口。数据库记录包含一个或多个标示记录唯一性的关键字段，以及一些用来表示记录所代表实体的特性的其他字段。

Python 字典可以被用作一种非持久化的数据库，因为可以使用字典键来创建、读取、更新或删除关联到给定字典键的值。这个值可以是一个字段元组或是一个记录。还缺少的就是在会话间存储数据的能力。字典作为数据库的概念已经被探索多年，并且已经存在各种形式的持久化字典。最久远的是文件的数据库管理(database management, DBM)家族。

3.1.1 使用 DBM 作为持久化字典

DBM 文件起源于 UNIX，但是多年以来也在其他平台中得到了发展。Python 支持多种变体。这些变体被隐藏在 dbm 模块下面。它会基于当前操作系统可用的库来自动确定最佳解决方案。如果没有找到原生 DBM 库，dbm 模块就会使用基本的 Python 版本的 DBM 库。

DBM 系统是字典的简化版本，因为键和值都必须是字符串。这意味着如果正在使用非字符串数据，就必须使用一些数据转换和字符串格式化。DBM 数据库的优势在于它易用、快速并相当简洁。

可以通过回顾第 2 章中工具借用的示例来了解 DBM 是如何工作的。在当时的示例中，你把电子表格作为主数据源。那么如果决定把解决方案迁移为纯粹的 Python 应用，该怎么办呢？你会需要一种存储机制来存储各种各样的数据元素。

前面介绍过，电子表格有两个工作表，一个表示能被租用的工具，另一个表示会员的实际租用。记录的格式如表 3-1 所示。

表 3-1 工具借用数据实体

TOOL	LOAN
ItemID	ItemID
Name	Name
Description	Description
Owner	Owner
Price	Borrower
Condition	Date Borrowed
Date Registered	Date Returned

这个设计对于人工操作数据表是很好的，但是如果想要把它转换成一个成熟的数据应用，需要克服存在的一些问题：

- 首先，在两个实体之间有很多冗余。Name、Description 和 Owner 字段都是冗余的，因此无论何时它们被修改，都要在两个地方进行。
- 两个实体都使用 ItemID 作为主键。这意味着 ItemID 既表示一个 Tool 又表示一个 Loan，这会令人困惑。
- 几个字段存储了服务订阅者的名字，但是最好有一个单独的实体用来描述那些成员，并从其他实体中引用这个成员实体。
- 最后，尽管一开始这是一个工具租借应用，但是没有理由把它限制为工具。成员也可以借书或 DVD 或任何其他东西。所以不能把它限制为工具，可以把 Tool 实体重命名为 Item。而为了与它保持一致，可以通过重命名应用来反映它更加通用的方法。把它称为 LendyDB。



注意：在此，我们删除了工具借出数据中的重复数据，并把它们分割为单独的条目。这些操作在数据设计过程中很常用，也被称为规范化。这是一个非常正式的原理。有很多书是关于这个主题的。虽然本书仅介绍主要原理，但它是良好数据库设计的重要组成部分。如果你需要设计一个高效的、大容量数据库，就应该研究规范化并熟悉这项技术。

通过很小的努力，可以重新整理这些东西来克服电子表格存在的问题。表 3-2 展示了数据库设计的结果。

现在你有了三个实体，所以需要把数据存储三个数据文件中。可以为此使用 DBM 格式，因为每个实体现在有唯一的标识字段。这个字符串格式的字段作为 DBM 键也工作得很好。需要为这些文件填充数据，而这意味着要重新格式化电子表格中的数据。可以编写一个 Python 程序来实现这一点，但是由于样本数据集很小，直接剪切并粘贴数据到新格式会更容易些(或者可以从下载网站的 Chapter3.zip 的 LendyDB 文件夹中提取文件)。一旦拥有数据，就可以相当轻松地把它保存在 DBM 文件中，就像在接下来的“试一试”中展示的那样。

表 3-2 LendyDB 数据设计

ITEM	MEMBER	LOAN
ItemID	MemberID	LoanID
Name	Name	ItemID
Description	Email	BorrowerID
OwnerID		Date Borrowed
Price		Date Returned

(续表)

ITEM	MEMBER	LOAN
Condition		
Date Registered		

试一试：创建一个 LendyDB DBM 数据库(create-lendyDB.py)

在这个“试一试”中，将把工具借出电子表格中的数据转换为 LendyDB 的数据格式，并把它保存为三组 DBM 文件。然后，通过读取文件并打印内容，你将证实它可以正确工作。完成下面的步骤：

(1) 创建一个项目目录并把它命名为 LendyDB。

(2) 启动你喜欢的编辑器或 IDE，然后输入下面的代码(或加载本书网站的 create-lendyDB.py 文件)：

```
import dbm

# ID, Name, Description, OwnerID, Price, Condition, DateRegistered
items = [
    ['1', 'Lawnmower', 'Tool', '1', '$150', 'Excellent', '2012 - 01 - 05'],
    ['2', 'Lawnmower', 'Tool', '2', '$370', 'Fair', '2012 - 04 - 01'],
    ['3', 'Bike', 'Vehicle', '3', '$200', 'Good', '2013 - 03 - 22'],
    ['4', 'Drill', 'Tool', '4', '$100', 'Good', '2013 - 10 - 28'],
    ['5', 'Scarifier', 'Tool', '5', '$200', 'Average', '2013 - 09 - 14'],
    ['6', 'Sprinkler', 'Tool', '1', '$80', 'Good', '2014 - 01 - 06']
]

# ID, Name, Email
members = [
    ['1', 'Fred', 'fred@lendylib.org'],
    ['2', 'Mike', 'mike@gmail.com'],
    ['3', 'Joe', 'joe@joesmail.com'],
    ['4', 'Rob', 'rjb@somcorp.com'],
    ['5', 'Anne', 'annie@bigbiz.com'],
]

# ID, ItemID, BorrowerID, DateBorrowed, DateReturned
loans = [
    ['1', '1', '3', '4/1/2012', '4/26/2012'],
    ['2', '2', '5', '9/5/2012', '1/5/2013'],
    ['3', '3', '4', '7/3/2013', '7/22/2013'],
    ['4', '4', '1', '11/19/2013', '11/29/2013'],
    ['5', '5', '2', '12/5/2013', 'None']
]

def createDB(data, dbName):
    try:
```

```

    db = dbm.open(dbName, 'c')
    for datum in data:
        db[datum[0]] = ','.join(datum)
    finally:
        db.close()
    print(dbName, 'created')

def readDB(dbName):
    try:
        db = dbm.open(dbName, 'r')
        print('Reading ', dbName)
        return [db[datum] for datum in db]
    finally:
        db.close()

def main():
    print('Creating data files...')
    createDB(items, 'itemdb')
    createDB(members, 'memberdb')
    createDB(loans, 'loandb')

    print('reading data files...')
    print(readDB('itemdb'))
    print(readDB('memberdb'))
    print(readDB('loandb'))

if __name__ == "__main__": main()

```

(3) 保存文件为 create-lendyDB.py 并运行它。确定你的输出看起来如下所示:

```

Creating data files...
itemdb created
memberdb created
loandb created
reading data files...
Reading itemdb
[b'2,Lawnmower,Tool,2,$370,Fair,2012-04-01', b'3,Bike,Vehicle,3,$200,
Good,2013-03-22', b'1,Lawnmower,Tool,1,$150,Excellent,2012-01-05',
 b'6,Sprinkler,Tool,1,$80,Good,2014-01-06', b'4,Drill,Tool,4,$100,
Good,2013-10-28', b'5,Scarifier,Tool,5,$200,Average,2013-09-14']
Reading memberdb
[b'2,Mike,mike@gmail.com', b'3,Joe,joe@joesmail.com', b'1,Fred,
fred@lendylib.org', b'4,Rob,rjb@somcorp.com', b'5,Anne,annie@bigbiz.com']
Reading loandb
[b'2,2,5,9/5/2012,1/5/2013', b'3,3,4,7/3/2013,7/22/2013', b'1,1,3,4/1/2012,
4/26/2012', b'4,4,1,11/19/2013,11/29/2013', b'5,5,2,12/5/2013,None']

```

(4) 检查 LendyDB 文件夹的内容。你应该看到三组文件，每组有三个文件。每组对应一个实体。组内文件的扩展名为 .bak、.dat 和 .dir。

示例说明

首先，导入了 dbm 模块(模块在内部分析你的系统，决定哪个 DBM 库是可用的并把它初始化以供使用)。然后通过从 Excel 电子表格数据中提取数值创建了原始数据项。注意，你已经更改了项的 Description 字段，所以它现在记录你拥有什么类型的东西(工具、书籍、DVD 等)。本可以创建一个额外的字段，而这也是同样有效的。但是对于这个练习，选择重用已存在的字段名。

然后定义了 createDB()函数。它以 c 模式打开了 DBM 数据库文件。c 代表创建模式(如果文件不存在，c 模式会创建一个新文件。如果已存在，它会打开这个文件)。然后使用一个 for 循环来读取每一个数据项并把它存储在数据库中，使用第一个字段作为主键，然后将所有字段连接成逗号分隔的字符串，这个字符串会作为值。

你使用了 try/finally 结构体来确保所有数据都被写入到文件中，并且文件被适当地关闭。

readDB()函数是 createDB()的相反操作。它使用 r 也就是读取模式打开了文件，然后返回了用列表推导产生的一个列表。如果预计数据库会非常大，也可以把这个函数变为一个生成器并轮流返回每一行。因为你不认为租借数据库会包含大量物品或成员，所以返回一个列表也不错。

最后，main()函数对每个数据实体都调用了一次 createDB()函数。注意，你并没有提供任何文件扩展名。dbm 自己做了这件事。然后 main()通过为每个数据库打印 readDB()的输出检查了数据是否被正确地创建。dbm 创建的数据库包含三个文件。一个文件保存了实际数据，其他两个文件保存着 dbm 用来查找数据文件中记录的索引信息。就是这个索引机制让 dbm 远远快于在纯文本文件中的简单顺序搜索。你不应该直接编辑 dbm 文件，因为这可能损坏你的数据库。



注意：用于 dbm 文件操作的模式字符串稍微有别于普通文件模式。默认的 r 模式是用来只读访问已存在的数据库。w 用来读取或写入已存在的数据库。c 创建新数据库或打开已存在的数据库。n 永远会创建一个新的空数据库。

创建好自己的数据库之后，现在可以使用它读取或编辑内容。这时 Python 的命令提示符会是很好的工具。所以从你保存数据文件的文件夹中启动 Python 解释器，并输入如下代码：

```
>>> import dbm
>>> items = dbm.open('itemdb')
>>> members = dbm.open('memberdb')
>>> loans = dbm.open('loandb', 'w')
>>> loan2 = loans['2'].decode()
>>> loan2
'2,2,5,9/5/2012,1/5/2013'
>>> loan2 = loan2.split(',')
>>>
```



```
>>> loan2
['2', '2', '5', '9/5/2012', '1/5/2013']
>>> item2 = items[loan2[1]].decode().split(',')
>>> item2
['2', 'Lawnmower', 'Tool', '2', '$370', 'Fair', '2012-04-01']
>>> member2 = members[loan2[2]].decode().split(',')
>>> member2
['5', 'Anne', 'annie@bigbiz.com']
>>> print('{} borrowed a {} on {}'.format(
... member2[1],item2[1],loan2[3]))
Anne borrowed a Lawnmower on 9/5/2012
```

通过之前的命令，打开了三个数据库，提取数字为 2 的借出(使用 `decode()` 把 dbm 字节格式转换为一个正常的 Python 字符串)，并把它分离为单独的域。然后通过使用借出记录值作为键，你提取了对应的成员和物品记录。最后，以可读的格式打印消息来报告数据。

当然，也可以在数据集中创建新记录。下面展示了如何创建一个新的借出记录：

```
>>> max(loans.keys()).decode()
'5'
>>> key = int(max(loans.keys()).decode()) + 1
>>> newloan = [str(key), '2', '1', '4/5/2014']
>>> loans[str(key)] = ','.join(newloan)
>>> loans[str(key)]
b'6,2,1,4/5/2014'
```

通过上面的代码，使用内置的 `max()` 函数在 `loans` 数据库中查找已存键的最高值。然后通过把这个最大值转换成一个整数值后加 1 创建了一个新键。然后，使用了新键值的字符串形式创建了一个新的借出记录。然后，使用新键的字段值把记录写出到数据库。最后，使用新键的值读回记录，以检查新记录是否存在。

可以看到，即使有多个条目，DBM 文件仍可以被用作数据库。然而，如果数据并不是自然的基于字符串或者有很多字段，那么提取字段并转换成合适的格式就变得非常乏味。可以编写辅助函数或方法完成这个转换。但还有一种更加轻松的方式。Python 有一个模块，可以把任意的 Python 对象存储到文件中，还可以在不需要做任何数据转换的情况下读回它们。是时候介绍一下 `pickle` 模块了。

3.1.2 使用 Pickle 存取对象

`pickle` 模块的目的是把 Python 对象转换成二进制字节序列。被转换的对象类型包括基本数据类型，例如整型和布尔值，以及系统和用户定义的类，甚至还包括容器，比如列表、元组和函数(除了那些使用 `lambda` 定义的)。可以被 `pickle` 操作的对象有一些限制条件。这些限制条件可以在模块文档中找到。

`pickle` 本身并不是一个数据管理解决方案。它仅仅把对象转换成二进制序列。这些序列可以被保存在二进制文件中，并再读回来。所以它们可以被用作一种数据持久性的形式。但是 `pickle` 并没有提供任何方式搜寻被存储的对象或从许多被存储的项中取回一个对象。

你必须读回整个存储库并访问对象。当只是想要保存一个程序的状态时, pickle 是很理想的。这样可以启动它并从之前同样的位置继续(例如, 你正在玩一个游戏)。



注意: 在计算机科学与技术中, 为了存储或网络传输把数据转换成字符串(或字节)是常见的操作。同样地, 这个过程有一个通用的名称: 序列化(有时也称为编组)。pickle 是 Python 特有的序列化形式。JavaScript 对象表示法(JSON)数据格式是另一种序列化形式。它被广泛地应用于跨语言环境中, 尤其是在网络中。你将在第 5 章看到更多关于 JSON 的知识。相比于 JSON, pickle 更加强大但是不如它通用。因为它仅局限于 Python 应用中。

pickle 模块提供了几个函数和类, 但是通常你只会使用 dump()和 load()函数。dump()函数把一个对象(或多个对象)转储到一个文件中。load()函数从一个文件(通常是之前用 dump 转储的对象)读取一个对象。

如果想看看这两个函数的工作原理, 可以使用交互式命令提示符来实验前面章节中的 LendyDB 的 Item 数据定义。首先创建一个项并使用元组而不是字符串来表示所有值:

```
>>> import pickle
>>> anItem = ['1', 'Lawnmower', 'Tool', '1', '$150', 'Excellent', '2012-01-05']
>>> with open('item.pickle', 'wb') as pf:
...     pickle.dump(anItem, pf)
...
>>> with open('item.pickle', 'rb') as pf:
...     itemCopy = pickle.load(pf)
...
>>> print(itemCopy)
['1', 'Lawnmower', 'Tool', '1', '$150', 'Excellent', '2012-01-05']
>>>
```

注意, 需要为 pickle 文件使用二进制文件模式。最需要注意的是, 你从文件中得到了一个列表而不仅仅是一个字符串。当然, 这些元素全都是字符串。所以有兴趣可以试试 pickle 一些不同的数据类型:

```
>>> funData = ('a string', True, 42, 3.14159, ['embedded', 'list'])
>>> with open('data.pickle', 'wb') as pf:
...     pickle.dump(funData, pf)
...
>>> with open('data.pickle', 'rb') as pf:
...     copyData = pickle.load(pf)
...
>>> print (copyData)
('a string', True, 42, 3.14159, ['embedded', 'list'])
>>>
```

显然，你存入的数据和取出的数据是相同的。唯一需要额外注意的是，pickle 并不安全。它会潜在地执行 unpickle 的对象。所以你应该永远不要用 pickle 去读取从非信任源获取的数据。但是对于在可控环境中的本地对象持久化，它简单而且高效。如果在自己的项目中使用 pickle，应该意识到你可能遇到一些 pickle 特有的异常。这样，你可能想要用 try/except 结构体来包装你的代码。

要对 LendyDB 项目使用 pickle，一个大问题是只能通过把整个文件读入内存来访问数据。如果可以对任意 Python 对象都有一个结合了 pickle 和 dbm 特性的索引文件，岂不是很好？其实这是可行的，shelve 模块已经为你实现了这一点。

3.1.3 使用 shelve 访问对象

shelve 模块结合了 dbm 模块对文件随机存取的能力和 pickle 模块序列化 Python 对象的能力。它并不是一个完美的解决方案，因为键仍然必须是字符串，并且 pickle 存在的安全问题也同样存在于 shelve 中。所以必须确保数据源是安全的。与 dbm 文件一样，模块不能告诉你修改数据是否被读入内存。所以你必须通过对同样的关键字再次赋值来显式地将任何修改写回到文件。最后，dbm 文件对于可以存储对象的大小有一些限制，并且它们也并不支持从比如多个线程或用户的并发读取。然而，对于很多项目来说，shelve 为存储和访问数据提供了一个简单、轻量级且快速的解决方案。

因此，就你看来，shelve 的行为就像一个字典。对于几乎要对字典做的所有事情，你都可以用 shelve 实例实现。唯一的区别就是数据仍然在硬盘中而不是在内存中。这显然会有速度上的差异。但是另一方面，这意味着可以在内存有限的情况下操作非常大的字典。

在使用 shelve 构建 LendyDB 之前，你会试试一些假设数据，这包括更多的数据类型甚至用户自定义的类。首先，你创建了一个 shelve 数据库文件(或者，它们有时被称为 shelf)：

```
>>> shelf = shelve.open('fundata.shelve', 'c')
```

open 函数与之前讨论的 dbm 版本的函数接受同样的参数。由于要创建一个新的 shelf，因此使用了模式 c。现在可以开始向 shelf 中添加记录：

```
>>> shelf['tuple'] = (1, 2, 'a', 'b', True, False)
>>> shelf['lists'] = [[1, 2, 3], [True, False], [3.14159, -66]]
```

利用这些命令，你保存了两个记录。每个记录的内容都混合了多个 Python 数据类型。shelve 非常乐意存储它们，不需要任何需要的数据转换。可以通过读回值来检查 shelve 已经保存的项：

```
>>> shelf['tuple']
(1, 2, 'a', 'b', True, False)
>>> shelf['lists']
[[1, 2, 3], [True, False], [3.14159, -66]]
```

为了永久保存数据的变化，需要调用 close(通常，会使用一个 try/finally 结构体，或不

同于 dbm，可以使用上下文管理器样式)：

```
>>> shelf.close()
>>> shelf['tuple']
Traceback (most recent call last):
  File "C:\Python33\lib\shelve.py", line 111, in __getitem__
    value = self.cache[key]
KeyError: 'tuple'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "C:\Python33\lib\shelve.py", line 113, in __getitem__
  f = BytesIO(self.dict[key.encode(self.keyencoding)])
File "C:\Python33\lib\shelve.py", line 70, in closed
  raise ValueError('invalid operation on closed shelf')
ValueError: invalid operation on closed shelf
>>>
```

可以看到，在关闭 shelf 之后，不能再访问数据了。需要重新打开 shelf。

现在可以尝试一些稍微复杂的东西了。首先，定义一个类，创建一些实例，然后把它存储到 shelf 中：

```
>>> class Test:
...     def __init__(self,x,y):
...         self.x = x
...         self.y = y
...     def show(self):
...         print(self.x, self.y)
...
>>> shelf = shelve.open('test.shelve','c')
>>> a = Test(1,2)
>>> a.show()
1 2
>>> b = Test('a','b')
>>> b.show()
a b
>>> shelf['12'] = a
>>> shelf['ab'] = b
```

到目前为止，一切顺利。你已经保存了类的两个实例。把它们取回来也非常容易：

```
>>> shelf['12']
<__main__.Test object at 0x01BD1570>
>>> shelf['ab']
<__main__.Test object at 0x01BD1650>
>>> c = shelf['12']
>>> c.show()
1 2
```

```
>>> d = shelf['ab']
>>> d.show()
a b
>>> shelf.close()
```

注意，返回的对象被称为 `__main__.Test` 对象。这会引发一个关于保存和恢复用户自定义类的非常重要的警告。你必须确保 `shelf` 用于保存的类定义也同样适用于从 `shelf` 读回类的模块，并且类定义必须一致。如果在写数据和读数据期间类定义发生了改变，结果将会变得不可预知。让类可见的常用做法是把它放入自己的模块。之后这个模块可以被导入，并在写入和读取 `shelf` 的代码中使用。



注意：可以在类中定义两个特殊方法(`__getstate__`和`__setstate__`)。这两个方法准确地告诉 `pickle`(和 `shelve`)哪些特性要保存。如果这两个方法本身没有改变，这可以避免当类定义发生改变时的一些问题。文档包含了这个机制的工作示例。总之，如果可能的话，最好避免改变类定义。

是时候再次访问你的借出库 `LendyDB` 了。这一次，使用 `shelve` 模块重复之前针对 `dbm` 数据库的操作。

试一试：使用 `shelve` 存储 `LendyDB`(`shelve-lendyDB.py`)

在这个“试一试”中，使用 `shelve` 重复了 `dbm` 示例中的功能。结果，代码要更加简单。为此，完成下面的步骤：

(1) 切换到 `LendyDB` 项目文件夹。

(2) 打开你最喜欢的编辑器或 IDE，并输入下面的代码(或加载下载文件中的 `LendyDB` 文件夹中的 `shelve-lendyDB.py` 文件)：

```
import shelve

# ID, Name, Description, OwnerID, Price, Condition, DateRegistered
items = [
    ['1', 'Lawnmower', 'Tool', '1', '$150', 'Excellent', '2012-01-05'],
    ['2', 'Lawnmower', 'Tool', '2', '$370', 'Fair', '2012-04-01'],
    ['3', 'Bike', 'Vehicle', '3', '$200', 'Good', '2013-03-22'],
    ['4', 'Drill', 'Tool', '4', '$100', 'Good', '2013-10-28'],
    ['5', 'Scarifier', 'Tool', '5', '$200', 'Average', '2013-09-14'],
    ['6', 'Sprinkler', 'Tool', '1', '$80', 'Good', '2014-01-06']
]

# ID, Name, Email
members = [
    ['1', 'Fred', 'fred@lendylib.org'],
    ['2', 'Mike', 'mike@gmail.com'],
    ['3', 'Joe', 'joe@joesmail.com'],
```

```
[
    ['4', 'Rob', 'rjb@somcorp.com'],
    ['5', 'Anne', 'annie@bigbiz.com'],
]

# ID, ItemID, BorrowerID, DateBorrowed, DateReturned
loans = [
    ['1', '1', '3', '4/1/2012', '4/26/2012'],
    ['2', '2', '5', '9/5/2012', '1/5/2013'],
    ['3', '3', '4', '7/3/2013', '7/22/2013'],
    ['4', '4', '1', '11/19/2013', '11/29/2013'],
    ['5', '5', '2', '12/5/2013', 'None']
]
```

```
def createDB(data, shelfname):
    try:
        shelf = shelve.open(shelfname, 'c')
        for datum in data:
            shelf[datum[0]] = datum
        finally:
            shelf.close()

def readDB(shelfname):
    try:
        shelf = shelve.open(shelfname, 'r')
        return [shelf[key] for key in shelf]
    finally:
        shelf.close()

def main():
    print('Creating data files...')
    createDB(items, 'itemshelf')
    createDB(members, 'membersshelf')
    createDB(loans, 'loansshelf')

    print('reading items...')
    print(readDB('itemshelf'))
    print('reading members...')
    print(readDB('membersshelf'))
    print('reading loans...')
    print(readDB('loansshelf'))

if __name__ == "__main__": main()
```

- (3) 保存文件为 shelve-lendyDB.py 并运行它。
- (4) 检查你的输出是否匹配下面的输出：

```
Creating data files...
reading items...
[['1', 'Lawnmower', 'Tool', '1', '$150', 'Excellent', '2012-01-05'], ['3', 'Bike',
', 'Vehicle', '3', '$200', 'Good', '2013-03-22'], ['2', 'Lawnmower', 'Tool', '2',
', '$370', 'Fair', '2012-04-01'], ['5', 'Scarifier', 'Tool', '5', '$200', 'Averag
```



```
e', '2013-09-14'], ['4', 'Drill', 'Tool', '4', '$100', 'Good', '2013-10-28'], ['6', 'Sprinkler', 'Tool', '1', '$80', 'Good', '2014-01-06']]
reading members...
[['1', 'Fred', 'fred@lendylib.org'], ['3', 'Joe', 'joe@joesmail.com'], ['2', 'Mike', 'mike@gmail.com'], ['5', 'Anne', 'annie@bigbiz.com'], ['4', 'Rob', 'rjb@somcorp.com']]
reading loans...
[['1', '1', '3', '4/1/2012', '4/26/2012'], ['3', '3', '4', '7/3/2013', '7/22/2013'], ['2', '2', '5', '9/5/2012', '1/5/2013'], ['5', '5', '2', '12/5/2013', 'None'], ['4', '4', '1', '11/19/2013', '11/29/2013']]
```

(5) 启动 Python 解释器并输入下面代码来试验数据:

```
>>> import shelve
<module 'shelve' from 'C:\\Python33\\lib\\shelve.py'>
>>> items = shelve.open('itemshelf', 'w')
>>> members = shelve.open('membershelf', 'w')
>>> loans = shelve.open('loanshelf', 'w')
>>> loan2 = loans['2']
>>> loan2
['2', '2', '5', '9/5/2012', '1/5/2013']
>>> item2 = items[loan2[1]]
>>> item2
['2', 'Lawnmower', 'Tool', '2', '$370', 'Fair', '2012-04-01']
>>> member2 = members[loan2[2]]
>>> print('{} borrowed a {} on {}'.format(
... member2[1], item2[1], loan2[3]))
Anne borrowed a Lawnmower on 9/5/2012
>>>
```

(6) 输入下面代码, 添加一个借出记录:

```
>>> key = int(max(loans.keys())) + 1
>>> newloan = [str(key), '2', '1', '4/5/2014']
>>> loans[str(key)] = newloan
>>> loans[str(key)]
['6', '2', '1', '4/5/2014']
>>> loans.close() # make the change permanent
```

示例说明

文件与使用 dbm 的文件非常相似。首先, 导入了 shelve 而不是 dbm。后面的三组数据定义与之前示例中的相同。然后定义了两个函数: createDB() 和 readDB()。

这两个函数是 shelve 帮助简化代码的地方。对于创建数据库, 函数打开 shelf, 然后直接把数据写入到 shelf 中, 并不需要使用字符串 join() 方法处理数据。对于读取数据库, 过程几乎是一模一样的, 但是使用了一个列表推导来取回、存储、返回 shelf 内容。

除了几个打印消息地方的微调, main() 函数也和 dbm 示例非常类似。

此时, shelve 解决方案看起来并没有什么巨大的优势。但是, 当开始访问数据库并修改它时, 情况就开始不同了。你打开了三个 shelf 并重复 dbm 部分的练习。但是这一次,

你不需要再进行 `split()` 操作来获得一个列表, 并且你不需要使用 `decode()` 把字节转换成普通字符串。这让代码变得更短也更易读(如果记录包含混合类型, 节省的工作会更加明显)。

最后, 创建了一个新的借出记录。同样, 这并不需要任何解码或字符串连接。当关闭 `shelf` 时, 要确保数据被写入磁盘。

现在你已经看到 Python 为存储和获取对象所提供的各种各样的选项。特别的, `shelve` 模块提供了一种简洁、快速、简单的持久化机制。如果有一种在内存中使用 Python 字典的解决方案, 切换到 `shelve` 解决方案是非常简单的。然而, 这对于复杂的数据处理还远远不够。类似于基于非键值查找一组记录或对数据排序等操作, 本质上仍然需要把数据全部读入内存。唯一可以避免的方法是转用一个成熟的数据库解决方案。然而, 在接触它之前, 你应该先了解一下 Python 为简化内存中数据集的数据分析所提供的一些帮助。

3.2 使用 Python 分析数据

面对一组数据, 你通常想要问一些关于它的问题。例如, 在借出库示例中, 你也许想要知道物品的总成本, 或甚至是单个物品的平均成本。你也许想要知道谁贡献的物品最多, 一定时间内哪些物品处于被借出状态, 等等。可以用 Python 做这个事情, 并且可以使用所有标准 Python 特性来编写函数以回答那些问题。然而, Python 有一些强大的特性。这些特性往往被忽视, 但是在分析数据集上非常有用。

在本节中, 你会看到一些可用的内置特性, 尤其是语言的函数式程序设计特性。然后, 你会将注意力转向 `itertools` 模块。相比于标准库模块, 这个模块提供了可以节省时间和计算资源的更高级特性。

3.2.1 使用 Python 的内置特性分析数据

当分析数据时, 选择正确的数据结构非常重要。例如, Python 包含一个 `set` 数据类型。它会自动去除重复元素。如果仅仅在意唯一值, 把数据转换(或抽出)到集合中可以大大地简化过程。同样的, 使用 Python 字典提供关键字访问而不是数值索引可以改善代码的可读性和可靠性(你在第 2 章已经看到一个这样的示例, 它对比了基于字典的 `CSV reader` 和标准的基于元组的 `reader`)。如果发现代码正变得复杂, 通常有必要停下来看看是否其他数据结构会有所帮助。

除了广泛的数据结构, Python 也提供许多可以使用的内置和标准库函数, 比如 `any`、`all`、`map`、`sorted` 和切片(切片在技术上并不是一个函数而是一个操作, 但是它也像一个函数一样返回一个值)。当使用 Python 生成器表达式和列表推导组合这些函数时, 你就有了一个强大的工具包, 用于切片和切块你的数据。

可以把这些技术应用到 `LendyDB` 数据, 从而回答在这节开头段落提出的问题。你现在就可以试试。

试一试：使用 Python 分析 LendyDB(lendydata.py)

在这个“试一试”中，会使用标准 Python 特性来回答之前提出的关于 LendyDB 数据的问题：1)所有物品的总成本是多少？2)单个物品的平均成本是多少？3)谁贡献的物品最多？4)一定时间内哪些物品处于被借出状态？为了回答这些问题，请完成下面的步骤。

(1) 创建一个名为 lendydata.py 的模块。它包含下面的代码(或者从下载文件的 Analysis 文件夹中加载它)：

```
items = [
    ['ID', 'Name', 'Description', 'OwnerID', 'Price', 'Condition', 'Registered'],
    ['1', 'Lawnmower', 'Tool', '1', '$150', 'Excellent', '2012-01-05'],
    ['2', 'Lawnmower', 'Tool', '2', '$370', 'Fair', '2012-04-01'],
    ['3', 'Bike', 'Vehicle', '3', '$200', 'Good', '2013-03-22'],
    ['4', 'Drill', 'Tool', '4', '$100', 'Good', '2013-10-28'],
    ['5', 'Scarifier', 'Tool', '5', '$200', 'Average', '2013-09-14'],
    ['6', 'Sprinkler', 'Tool', '1', '$80', 'Good', '2014-01-06']
]

members = [
    ['ID', 'Name', 'Email'],
    ['1', 'Fred', 'fred@lendylib.org'],
    ['2', 'Mike', 'mike@gmail.com'],
    ['3', 'Joe', 'joe@joesmail.com'],
    ['4', 'Rob', 'rjb@somcorp.com'],
    ['5', 'Anne', 'annie@bigbiz.com'],
]

loans = [
    ['ID', 'ItemID', 'BorrowerID', 'DateBorrowed', 'DateReturned'],
    ['1', '1', '3', '4/1/2012', '4/26/2012'],
    ['2', '2', '5', '9/5/2012', '1/5/2013'],
    ['3', '3', '4', '7/3/2013', '7/22/2013'],
    ['4', '4', '1', '11/19/2013', '11/29/2013'],
    ['5', '5', '2', '12/5/2013', 'None']
]
```

(2) 启动 Python 解释器并使用下面的命令导入数据：

```
>>> from lendydata import *
```

(3) 为了回答问题“所有物品的总成本是多少？”，输入下面的代码：

```
>>> def cost(item):
...     return int(item[4][1:])
...
>>> cost(items[2])
370
>>> sum(cost(item) for item in items[1:])
1100
```


(4) 为了回答问题“单个物品的平均成本是多少？”，输入下面的代码：

```
>>> sum(cost(item) for item in items[1:])/len(items) - 1
183.333333333334
>>>
```

(5) 为了回答问题“谁贡献的物品最多？”，输入下面的代码：

```
>>> def owner(item): return item[3]
...
>>> for member in members[1:]:
...     count = 0
...     for item in items[1:]:
...         if owner(item) == member[0]:
...             count += 1
...         print(member[1],':',count)
...
Fred : 2
Mike : 1
Joe : 1
Rob : 1
Anne : 1
>>>
```

(6) 为了回答问题“一定时间内哪些物品处于被借出状态？”，输入下面的代码：

```
>>> def onLoan(loan): return loan[-1] == 'None'
...
>>> [items[int(loan[1])] for loan in loans if onLoan(loan)]
[['5', 'Scarifier', 'Tool', '5', '$200', 'Average', '2013-09-14']]
>>>
```

示例说明

首先，你创建了一个 Python 模块。它包含了你的样本数据并把数据导入到解释器中。注意一个小技巧：使用每个数据部分的第一个条目(索引为 0)来存储这部分的字段描述。

这两个有用的效果：

- 任何给定数据部分(三个部分)的每一个 ID 值现在匹配这部分相同行的相对于 0 的索引。例如，members 数据部分的“Mike”行的 ID 为 3，它现在可以通过 members[3] 被访问。
- 在程序和解释器中把它作为备忘录，你已经通过访问第一个记录访问到字段名。

缺点是在你处理的代码中，为了处理每部分行首记录，你必须记得把数据部分的长度和索引调整 1。

然后使用标准 Python 工具来回答关于数据的几个问题。对于每个问题，你定义了一个小的辅助函数。它一般只是从数据条目中删除一个字段。对于第一个问题，它通过抽取字符串值并在前面去除美元符号后转换成整型，最后把这个整型当作成本返回。然后你对一个生成器表达式使用了 Python 内置的 sum() 函数来计算物品的总成本。最后通过把总成本

除以物品数量从而计算出物品的平均成本。

为了找到谁贡献了哪些物品，你定义了 `owner()` 函数。它只是从一个物品记录中抽取 `ownerID` 字段。然后循环所有成员来检查每个成员拥有多少物品。

最后，创建了 `onLoan()` 辅助函数来决定哪些物品被借出了。这个函数根据 `DateReturned` 字段值是否为 `None` 返回一个布尔结果。然后你在列表推导中把 `onLoan()` 函数作为过滤条件使用。

在之前的“试一试”中，你看到了可以把内置函数和数据结构与循环和生成器结合起来回答大多数关于数据的问题。但有一个问题是，对于大量数据，这种技术需要在内存中存储大列表，而且可能会涉及多次循环那些列表。这可能变得非常慢而且很耗费资源。Python `itertools` 模块提供了几个可以大大减轻负担的函数。

3.2.2 使用 `itertools` 分析数据

标准 Python 库的 `itertools` 模块提供了一组工具。这些工具使用函数式编程原理接受可迭代对象并产生其他可迭代对象作为结果。这意味着可以将函数组合起来创建复杂的数据过滤器。

在了解 `itertools` 如何被用在 `LendyDB` 数据之前，你应该通过简单点的数据集了解其中的一些函数。这些函数非常强大，但是操作方法和你之前接触过的函数有一些不同。它们本质上都是处理迭代器的。你应该回想一下所有的标准 Python 集合和对象，比如文件，都是迭代器。也可以通过定义一些符合 Python 迭代器协议的方法来创建自定义迭代器。最简单的迭代器是字符串。文档主要用它来展示 `itertools` 函数。但是切记，这些函数可以用来处理任何迭代器，而不仅仅是字符串。

1. 工具函数

你首先看到的一组函数是相对简单的函数，主要用于为模块中的其他函数提供输入。`count()` 函数与内置的 `range()` 函数的工作原理非常相似。但 `range()` 产生有限的数字而 `count()` 从开始点产生无限数字序列。增加的步长可以通过可选参数 `stepsize` 控制。代码如下：

```
>>> import itertools as it
>>> for n in it.count(15,2) :
...     if n < 40: print(n, end=' ')
...     else: break
...
15 17 19 21 23 25 27 29 31 33 35 37 39
```

`repeat()` 函数要更加简单。它只是持续地或按照指定的次数重复它的参数。如下代码所示：

```
>>> for n in range(7):
...     print(next(it.repeat('yes ')), end='')
```

```
...
yes yes yes yes yes yes yes >>>
>>> list(it.repeat(6,3))
[6, 6, 6]
>>>
```



注意：几个 `itertools` 函数可能会产生一个无限的输出数据序列。这有可能会锁住程序并进入无限循环。需要格外小心，确保在使用这些函数时提供一个退出机制。

`cycle()` 函数会反复不断地在输入序列上轮转。这对于为负载均衡或资源分配创建轮式迭代是非常有用的。考虑一下这种情况：当有一些资源并想要把数据轮流分配给这些资源时。可以创建一个资源列表，然后轮转这个列表直到你分配完数据。可以使用列表作为资源来模拟这种技术，如下代码所示：

```
>>> res1 = []
>>> res2 = []
>>> res3 = []
>>> resources = it.cycle([res1,res2,res3])
>>> for n in range(30):
...     res = next(resources)
...     res.append(n)
...
>>> res1
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>> res2
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28]
>>> res3
[2, 5, 8, 11, 14, 17, 20, 23, 26, 29]
>>>
```

`chain()` 函数把所有的输入参数连接成一个列表，然后返回每一个元素。如果参数的类型都相同，则可以通过使用加操作符把集合连接起来达到同样的结果。但是 `chain()` 也可以在不兼容加操作符的容器类型上工作。以下是一个使用列表、字符串和集的示例：

```
>>> items = it.chain([1,2,3], 'astring', {'a', 'set', 'of', 'strings'})
>>> for item in items:
...     print(item)
...
1
2
3
a
s
t
```



```

r
i
n
g
a
of
set
strings

```

最后，还有一个 `islice()` 函数。它的工作原理与切片操作符类似。由于它使用了生成器，因此在内存使用上更加高效。它与普通切片有一个很大的区别：你不能使用负索引值来从末端倒数，因为迭代器并非都有定义完善的端点。

`islice()` 的使用方法如下：

```

>>> data = list(range(20))
>>> data[3:12:2]
[3, 5, 7, 9, 11]
>>> for d in it.islice(data,3,12,2): print(d, end=' ')
...
3 5 7 9 11

```

除了生成数据，`itertools` 可以做更多的事情。它也可以使用各种各样的数据处理函数来帮助分析数据。

2. 数据处理函数

`itertools` 有很多数据处理函数。它们或者接受输入数据并转换元素，或者以某种方式过滤内容。通过组合这些函数，可以创建复杂的数据处理工具。这些函数的一个共同特性是它们接受一个函数对象作为一个参数。



注意：在函数式编程风格中，传递函数作为参数是一个常见特性，但一开始看起来可能有些奇怪。你只需要记住，在 Python 中，函数也是对象。函数名和任意其他变量一样。它仅仅是函数对象的引用。同样的，可以传送一个函数名，比如 `f`，到另一个函数中，假设是 `g`。函数 `g` 可以在内部调用输入函数 `f`。返回一个布尔结果的函数通常被称为谓词(predicate)。

`compress()` 函数就像你在第 1 章和第 2 章中看到的位掩码的高级版本。它接受一个数据集作为第一个参数，一个布尔值集合作为第二个参数。返回第一个集合中对应着第二个集合 `True` 值的那些项。以下是一个简单的示例：

```

>>> for item in it.compress([1,2,3,4,5],[False,True,False,0,1]):
...     print(item)
...

```

2
5

注意,布尔值并不必是纯正的布尔值。它们可以是任何 Python 能转换成布尔值的东西,甚至是表达式。(itertools.filterfalse()函数的工作原理完全相同,但是是相反的。它返回那些对应布尔值为 False 而不是 True 的元素)。

同样地,dropwhile()和 takewhile()函数也有相关的但相反的效果。两个函数都接受一个输入函数和一个集合或迭代器作为参数,然后每次对输入数据元素应用函数。dropwhile()函数会忽略所有输入元素直到函数参数返回结果为 False。而 takewhile()则返回元素直到结果为 False。可以在这些示例中看到区别。它们使用同样的输入数据和参数函数:

```
>>> def singleDigit(n): return n < 10
...
>>> for n in it.dropwhile(singleDigit,range(20)): print(n,end=' ')
...
10 11 12 13 14 15 16 17 18 19
>>> for n in it.takewhile(singleDigit,range(20)): print(n,end=' ')
...
0 1 2 3 4 5 6 7 8 9
```

注意,这两个函数在第一次检测到触发器之后都停止处理数据。

看看下面这个示例:

```
>>> for n in it.dropwhile(singleDigit,[1,2,12,4,20,7,999]): print(n,end=' ')
...
12 4 20 7 999
```

注意,输出中包含了首个非个位数,但是后面却跟着个位数。原因如刚才所述:一旦 dropwhile 停止丢掉,之后就不会丢掉任何东西了(而 takewhile 的接受行为是类似的)。

accumulate()函数把它的输入函数应用到输入数据的每个元素和之前操作的结果(默认函数是加,第一个结果总是第一个元素)。因此,对于输入数据集[1,2,3,4]来说,初始值 result1 是 1。之后函数会被应用到 result1 和 2 产生 result2,再被应用到 result2 和 3 产生 result3,再被应用到 result3 和 4 产生 result4。输出是 result1, result2, result3 和 result4(最终结果值与应用 functools 模块的 reduce()函数相同)。以下是一个使用 accumulate()的默认加法操作符的示例:

```
>>> for n in it.accumulate([1,2,3,4]): print(n, end=' ')
...
1 3 6 10
```

3. 掌握 groupby()

groupby()是 itertools 函数中最有用、最强大的函数之一。但是它的一些小缺点容易让你犯错。它的基本功能是把输入数据根据输入函数提供的关键字收集到不同的组中,并根据自身因素把那些组作为迭代器返回。

第一个问题是，只要找到相同的关键字，函数就会分组。但是如果找到一个新关键字，它会创建一个新组。然后，如果在序列中找到原来的关键字，它会用同样的关键字创建一个新组而不是把新元素添加到原来的组中。为了避免这个行为，最好把输入数据用 `groupby()` 和相同的关键字函数排序。

第二个问题是，`groupby()`产生的组并不是真实的独立迭代器。它们实际上是原始输入集合的视图。因此，如果函数移到下一组数据，之前的组就会失效。为了之后处理而保留这些组的唯一办法是把它们复制到一个单独的容器中，比如列表。

为了巩固这些概念，你会看到一个示例。它产生了一组可以被独立处理的数据组。示例从一个最初的、不成熟的、但有缺陷的解决方案开始，逐步成为最终正确的解决方案。

首先，定义几组数据并使用内置的 `all()` 函数作为关键字。当所有输入数据项都为 `True` 时，`all()` 函数返回 `True`。

```
>>> data = [[1,2,3,4,5],[6,7,8,9,0],[0,2,4,6,8],[1,3,5,7,9]]
>>> for d in data: print(all(d))
...
True
False
False
True
```

然后，对数据应用 `groupby()` 函数：

```
>>> for ky,grp in it.groupby(data,key=all):
...     print(ky, grp)
...
True <itertools._grouper object at 0x7fd3ee2c>
False <itertools._grouper object at 0x7fd3ee8c>
True <itertools._grouper object at 0x7fd3ee2c>
```

可以看到，`groupby()` 返回了两个单独的组。它们的关键字都是 `True`。为了避免它，必须在应用 `groupby()` 之前对数据进行排序，如下：

```
>>> for ky,grp in it.groupby(sorted(data,key=all), key=all):
...     print(ky, grp)
...
False <itertools._grouper object at 0x7fd3ef4c>
True <itertools._grouper object at 0x7fd3ee2c>
```

现在，你想要尝试访问这些组，所以将每个组存储在一个变量中：

```
>>> for ky,grp in it.groupby(sorted(data,key=all), key=all):
...     if ky: trueset = grp
...     else: falseset=grp
...
>>> for item in falseset: print(item)
...
>>>
```


如你所见, `falseset` 为空。这是因为 `falseset` 组被最先创建, 然后后面的迭代器(`grp`)继续前进。这使刚刚存储在 `falseset` 中的值失效了。为了保存数据集方便后面处理, 需要将它们存为列表, 如下:

```
>>> groups = {True:[], False:[]}
>>> for ky,grp in it.groupby(sorted(data,key=all), key=all):
...     groups[ky].append(list(grp))
...
>>> groups
{False: [[[6, 7, 8, 9, 0], [0, 2, 4, 6, 8]]],
 True: [[[1, 2, 3, 4, 5], [1, 3, 5, 7, 9]]]}
>>>
```

注意, 你创建了一个字典。它的键就是所期待的那些(`True` 和 `False`), 而它们对应的值是列表。然后当找到它们时, 需要添加转换成列表的组。这可能看起来很复杂, 但是如果记得先对输入数据进行排序, 然后把 `groupby()` 产生的组复制为列表, 你会发现 `groupby()` 是一个强大而有用的工具。

3.2.3 使用 `itertools` 分析 `LendyDB` 数据

你已经看到 `itertools` 所提供的功能了, 所以现在是时候用 `LendyDB` 数据试一试它了。你希望重复标准工具所做的分析, 但是也要看看 `itertools` 函数能够有哪些帮助。记住, `itertools` 模块的真正作用并不是它可以提供一些新特性, 而是它让你能够更高效地处理大量数据。考虑到本章所用的小量数据, 实际上你不会看到任何效率上的改进。但是当数据量变大时, 它就会有区别了。

试一试: 使用 `itertools` 分析 `LendyDB` 数据

在这个“试一试”中, 会使用一些 `itertools` 函数来重复之前 `LendyDB` 数据的分析。为此, 完成以下这些步骤:

- (1) 切换到 `lendydata.py` 文件所在的文件夹。
- (2) 启动 Python 解释器并使用下面的命令导入数据文件:

```
>>> from lendydata import *
```

- (3) 导入 `itertools` 模块:

```
>>> from itertools import *
```

- (4) 为了回答问题“所有物品的总成本是多少?”, 输入下面的代码:

```
>>> def cost(item):
...     return int(item[4][1:])
...
>>> for n in islice(accumulate(cost(item) for item in items[1:]),
...                 len(items) - 2, None):
...     print(n)
```

```
...
1100
```

(5) 为了回答问题“单个物品的平均成本是多少？”，输入下面的代码：

```
>>> n/len(items) - 1
183.33333333334
>>>
```

(6) 为了回答问题“谁贡献的物品最多？”，输入下面的代码：

```
>>> def owner(item): return item[3]
...
>>> owners = {}
>>> for ky,grp in groupby(sorted(items[1:], key=owner), key=owner):
...     owners[ky] = len(list(grp))
...
>>> for member in members[1:]:
...     print(member[1], ' : ', owners[member[0]])
...
Fred : 2
Mike : 1
Joe : 1
Rob : 1
Anne : 1
```

(7) 为了回答问题“一定时间内哪些物品处于被借出状态？”，输入下面的代码：

```
>>> def returned(loan): return not (loan[-1] == 'None')
...
>>> [items[int(loan[1])] for loan in filterfalse(returned,loans)]
[['5', 'Scarifier', 'Tool', '5', '$200', 'Average', '2013-09-14']]
```

示例说明

和前一个“试一试”一样，创建了一些小的辅助函数来改善代码的可读性。为了回答第一个问题，使用 `accumulate()` 函数产生了连续的成本计数，然后使用 `islice()` 并指定一个开始索引 `len(items)-2` 和结束索引 `None` 来只抽取最后一项(为了考虑 `items` 开始的标题行，必须减 2)。因为结果 `n` 仍然在范围内，所以可以简单地用 `n` 除以物品个数来计算平均数。

相比于之前的“试一试”，对于谁贡献的物品最多这个问题的回答是不同的。因为你使用 `groupby()` 来收集相关的物品。在这种情况下，你只对组的大小感兴趣。所以使用 `len()` 来计算组的大小。然后使用传统风格迭代 `members` 来打印名称和计数。

在回答最后一个问题时，你反转辅助函数 `returned()` 的逻辑，让它返回一个物品是否已经被返回。通过在 `filterfalse()` 函数中使用它，可以发现那些还没有被返回的物品，也就是处于借出状态的物品。

在本节中，你已经了解了如何通过把传统的 Python 数据结构、函数和操作符，和 `itertools` 模块的函数式编程技术结合起来。这个组合可以帮助你较大数据集执行复杂的分析。然

而，当数据的量和复杂度达到一个临界点时，就需要另外一种方法。这也意味着将要介绍一种新技术：由结构化查询语言(Structured Query Language, SQL)驱动的关系型数据库。

3.3 使用 SQL 管理数据

本节会介绍一些有关 SQL 和关系型数据库的概念。你会了解如何使用 SQL 创建数据表并用数据填充它们，以及如何操纵表中的数据。接下来，你会链接数据表来获取数据间的关系。最后，将把这些技术应用到借出库数据。

3.3.1 关系型数据库的概念

关系型数据库的基本原理非常简单。它只是一组二维的表(tables)集合。列被称为字段(fields)而行被称为记录(records)。字段值可以指向其他记录，或者在同一个表中，或者在另一个表中。这就是关系部分。

一个拥有关于员工数据的表可能看起来如表 3-3 所示。

表 3-3 员工数据

EMPID	NAME	HIREDATE	GRADE	MANAGERID
1020304	John Brown	20030623	Foreman	1020311
1020305	Fred Smith	20040302	Laborer	1020304
1020307	Anne Jones	19991125	Laborer	1020304

注意这个数据展示出来的两个惯例：

- 有一个身份(ID)字段来唯一地标识每行。这个 ID 也被称为主键(primary key)。也可能有其他键。但按照惯例，几乎总有一个 ID 字段来唯一地标识一个记录。例如，当一个员工决定修改她的名字时，这很有帮助。
- 可以通过让一个字段保存另一个行的主键值将一行链接到另一行。因此，一个员工的管理者是通过 ManagerID 字段标识的。它只是指向同一个表中的另一个 EmpID。看看你的数据，可以看到 Fred 和 Anne 的管理者都是 John。反过来，John 也被其他人管理。但这个人的详细信息在表的这部分不可见。

关系基数

数据库的关系把两个或更多条目链接在一起。关系中涉及实体的个数被称为基数。关系可以是一对一。一个记录恰好链接到另一个记录。也可以是一对多，例如示例中的员工-管理者关系。

关系也可以是多对多。有个示例可以最好地解释这种关系。假如你有一个任务表。每个任务可以分配多个员工。同时，每个员工可以有多个任务。因此，员工和任务之间存在多对多关系。

任何数据库应用的大部分工作集中在数据库内部多对多关系的维护上。有几种形式的图形表示法(也称为实体关系图)被用来描述数据库结构。大多数图形表示法都侧重于表示每个关系的基数。

并不局限于在一个表中链接数据。可以为 Salary 创建另一个表。一个 salary 可以被关联到 Grade，而可以得到像表 3-4 一样的第二个表。

表 3-4 薪水数据

SALARYID	GRADE	AMOUNT
000010	Foreman	60000
000011	Laborer	35000

为了得到员工 John Brown 的薪水，首先在主员工数据表中查找 John 的级别。然后会在 Salary 表查询拥有这个级别的员工薪水。这样，就可以看到 John 作为领班的薪水为 \$60000。

关系型数据库因在关系中链接表的行的能力而得名。其他数据库类型包括网络数据库、层次数据库和平面文件数据库(这包括在本章之前介绍的 DBM 数据库)。对于大量数据，关系型数据库是迄今为止最常用的。

也可以做更加复杂的查询。在接下来的几节中将介绍这一点。但是在可以查询任何东西前，需要先创建一个数据库并插入一些数据。

3.3.2 结构化查询语言

结构化查询语言，或 SQL(发音为 Sequel 或字母 S-Q-L)，是操纵关系型数据库的标准软件工具。在 SQL 中，表达式通常被称为查询(query)，不管它是否真的返回任何数据。

SQL 由两部分组成。第一部分是数据定义语言(data definition language, DDL)。这是一组用来创建和修改数据库本身框架(它的结构)的命令。DDL 在不同的数据库中是不同的。每个供应商的 DDL 都有些微的语义差别。

SQL 的另一个部分是数据操作语言(data manipulation language, DML)。DML 在数据库中间要更加高度标准化。它被用来操纵数据库内容而不是结构。大部分时间内你都在使用 DML 而不是 DDL。



注意：在本书中使用的是 SQLite 数据库系统。它有三大优势。第一，它有一个作为 Python 标准库一部分的 API 模块。第二，它作为 SQL 的一个分支学起来很简单。第三，SQLite 基于一个单独的数据文件和代码库工作，所以你不需搭建一个数据库服务器，或担心任何与维护关系型数据库有关的数据管理任务。

你只需要简略浏览一下 DDL，只要能够创建(使用 CREATE 命令)和删除(使用 DROP 命令)数据库表就可以了。这样，可以继续填充数据，并以感兴趣的方式获取数据，甚至使用 DML 命令(INSERT, SELECT, UPDATE 和 DELETE)修改它。



注意: 为了使用 SQL 交互式提示符, 需要下载 SQLite 解释器, 因为 Python 没有自带它。可以在 SQLite 网站 <http://www.sqlite.org/download.html> 上找到它。版本可能不会精确地匹配, 但是数据库格式足够稳定。尽管版本可能没有精确地匹配, 但是 Python 的 sqlite3 模块基本上可以非常容易地使用最新的解释器。只需要为操作系统的 shell 程序下载二进制文件即可。Python 已经安装了库 (Linux 用户可以在包管理器中找到 SQLite。如果能找到的话, 这是安装它的最好方式)。

可以在 <http://sqlite.org/cli.html> 上找到关于 SQLite 解释器的官方指南。

1. 创建表

如果想要在 SQL 中创建一个表, 可以使用 CREATE 命令。它很容易使用, 格式如下所示:

```
CREATE TABLE tablename (fieldName, fieldName,...);
```



注意: 在解释器中, SQL 语句必须以一个分号结束。这是因为 SQL 语句可以跨越多行, 所以需要告诉解析器你什么时候完成。Python 代码执行的 SQL 是作为一个单独的、完整的字符串, 所以结束分号是不必要的。

与 Python 不同, SQL 是不区分大小写的, 也不在意空格或缩进级别。一种非正式的习惯被使用, 但并非必须严格遵守它, 而 SQL 本身一点也不在乎。

试着在 SQLite 中创建 Employee 和 Salary 表。首先要做的事情就是启动解释器。只需要使用一个命令行参数, 也就是数据库文件名, 来启动 sqlite3 即可。如果数据库文件存在, 解释器会打开它, 否则它会用那个名称创建一个新的数据库文件(如果彻底省略数据库文件名, 解释器仍然会处理你的命令, 但是你的数据仅会存在于 RAM 中, 并且当退出解释器时会不可挽回地消失)。

因此, 为了创建一个员工数据库, 可以按如下命令执行 SQLite 解释器:

```
$ sqlite3 employee.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
```

```
Enter SQL statements terminated with a ";"
sqlite>
```

解释器创建了一个名为 `employee.db` 的空数据库。在 `sqlite>` 提示符准备输入 SQL 命令。现在准备好创建一些表：

```
sqlite> create table Employee
...> (EmpID,Name,HireDate,Grade,ManagerID);
sqlite> create table Salary
...> (SalaryID, Grade,Amount);
sqlite> .tables
Employee Salary
sqlite>
```

注意，你把字段列表移到了单独的一行。这样看起来更加方便。列出的字段除了名称没有其他定义信息，比如数据类型。这是 SQLite 的一个特性。大多数数据库需要你指定名称以及类型。在 SQLite 中指定类型也是可能的，但这不是很重要(你会在本章之后看到更多的细节)。

也要注意，你通过使用 `.tables` 命令列出数据库中的表格来测试创建语句。SQLite 解释器支持数个这些用来了解你的数据库的点命令。`.help` 提供了一个命令列表以及关于它们功能的简要描述。

当创建一个表后，可以做很多其他的事情。除了为每列的数据声明类型，也可以为值指定约束条件。约束条件就是规则。数据库强制使用这些规则来保持数据的一致性。例如，`NOT NULL` 意味着必须为字段提供值，而 `UNIQUE` 意味着不能有其他记录的这个字段有相同的值。通常，你指定主键字段为 `NOT NULL` 和 `UNIQUE`。也可以指定哪个字段是 `PRIMARY KEY`。你会在本章后面看到这些更高级的创建选项。



注意：SQLite 提供了可以使用的最普通的限制条件集。一些商业数据库提供非常丰富并强大的限制选项。它试图通过限制条件来实施大部分关于数据的商业逻辑。这通常是一个错误，因为它强制你以一种特定的顺序插入或修改你的数据。这可能会变得越来越难解决。最好用约束条件来控制数据的完整性，并在程序代码中构建应用逻辑。

现在，暂且将最基本的表定义放在一边，开始接触关于实际创建一些数据的更有趣的话题。

2. 插入数据

在创建表之后的第一件事就是填充数据。使用 SQL `INSERT` 语句来实现。结构非常简单：


```
INSERT INTO Tablename ( column1, column2... ) VALUES ( value1, value2... );
```

另一种 INSERT 形式使用一个查询从数据库的其他地方选择数据，但是在现阶段太高级。可以在 SQLite 指南中读到它，在 <http://sqlite.org/lang.html> 上找到该指南。

执行下面命令，插入一些行到 Employee 表中：

```
sqlite> insert into Employee (EmpID, Name, HireDate, Grade, ManagerID)
...> values ('1020304', 'John Brown', '20030623', 'Foreman', '1020311');
sqlite> insert into Employee (EmpID, Name, HireDate, Grade, ManagerID)
...> values ('1020305', 'Fred Smith', '20040302', 'Laborer', '1020304');
sqlite> insert into Employee (EmpID, Name, HireDate, Grade, ManagerID)
...> values ('1020307', 'Anne Jones', '19991125', 'Laborer', '1020304');
```

也为 Salary 表插入一些行：

```
sqlite> insert into Salary (SalaryID, Grade, Amount)
...> values ('000010', 'Foreman', 60000);
sqlite> insert into Salary (SalaryID, Grade, Amount)
...> values ('000011', 'Laborer', 35000);
```

就这么简单。你已经创建了两个表并使用与介绍中描述的值一致的数据填充它们。注意，你为薪水数量使用了实际的数字，不仅仅是字符串表达式。SQLite 试图基于我们提供的 INSERT 输入值判断出正确的数据类型。因为让 SQLite 把薪水数据作为一个数值类型最有意义，你理所当然地通过指定薪水数据为数值而不是字符串格式将你的 INSERT 语句偏好告知 SQLite。

现在你已经准备好开始实验数据了。好戏开始了！

3. 读取数据

你使用 SQL 的 SELECT 命令从数据库读取数据。SELECT 是 SQL 的绝对核心，并且在所有 SQL 命令中它有最复杂的结构。首先你会看到最基本的形式，然后随着深入会添加额外的特性。最基本的 SELECT 语句如下：

```
SELECT column1, column2... FROM table1, table2...;
```



注意：可以使用特殊的通配符(*)而不是字段名列表来返回所有字段。你应该只在交互式提示符中工作时使用这个。如果是在应用代码中使用它，并且其他人为数据库添加了一个额外字段，你的程序就会失败。通过指定确切的返回字段，你的代码对数据库变动会有更大弹性。

为了得到数据库中所有员工的名字，使用下列代码：

```
sqlite> SELECT Name from Employee;
```

```
John Brown
Fred Smith
Anne Jones
```

得到了 `Employee` 表中的所有名字列表。这个情况下只有三个，但是如果有一个大数据库，可能又会比你想要的更多信息。为了限制输出，需要稍微限制搜索条件。为此，SQL 允许你添加一个 `WHERE` 子句到 `SELECT` 语句中，如下所示：

```
SELECT col1,col2... FROM table1,table2... WHERE condition;
```

`condition` 是一个任意复杂的布尔表达式。它内部甚至可以包含嵌套的 `SELECT` 语句。现在，添加一个 `WHERE` 子句来完善名字的搜索。这一次，只看员工名字：

```
sqlite> SELECT Name
...> FROM Employee
...> WHERE Employee.Grade = 'Laborer';
Fred Smith
Anne Jones
```

你只得到了两个名字，而不是三个(因为 John Brown 不是员工)。可以使用布尔操作符，比如 `AND`、`OR`、`NOT` 等，来扩展 `WHERE` 条件。注意，`=` 在一个 `WHERE` 条件中执行一个区分大小写的测试。当使用 `=` 测试，字符串大小写很重要。测试 `'laborer'` 就不会正常工作了！

SQLite 有一些可以被用来操纵字符串的函数。但它也有一个名为 `LIKE` 的比较操作符。该操作符使用 `%` 作为通配符实现更加灵活的搜索。刚才的示例用 `LIKE` 写出来如下所示：

```
sqlite> SELECT Name FROM Employee
...> WHERE lower(employee.grade) LIKE 'lab%';
Fred Smith
Anne Jones
```

在把 `grade` 转换成小写之后，你用字首的 `'lab'` 子字符串测试了它。当连同 `lower()`、`upper()` 和 SQLite 的其他字符串操纵函数一起使用时，`LIKE` 可以极大地增加基于文本搜索的范围。SQLite 文档有一个完整的可用函数列表。



注意：可以使用下面的技术在解释器中测试 SQLite 函数。在一个 `SELECT` 语句中，返回值可以是任意表达式，而表子句可以为空。通过组合这两个特性，可以编写像 `SELECT lower('FREDDY')` 这样的代码。而 SQLite 会返回值 `'freddy'`。当想要试验函数看它做些什么时，这非常有用。

需要注意的还有，在 `WHERE` 子句中使用了点符号(`employee.grade`)来表示 `Grade` 字段。在这个情况下是没有必要的，因为你仅仅在一个表(在 `FROM` 子句中指定的 `Employee`)上工

作。但是，当指定多个表时，需要解释清楚字段属于哪个表。例如，将查询改为查找所有薪水高于\$50000 员工的名字。需要考虑两个表中的数据：

```
sqlite> SELECT Name, Amount
...> FROM Salary, Employee
...> WHERE Employee.Grade = Salary.Grade
...> AND Salary.Amount > 50000;
John Brown|60000
```

不出所料，你只得到了一个名字，也就是领班的名字。但是请注意，你也得到了薪水，因为你在选择列的列表中添加了 Amount。同时注意 WHERE 子句有两部分，它们通过一个 AND 布尔操作符连接在一起。第一部分通过确保共同字段相等来链接两个表。这在 sql 中被称为 join。在这个查询中，还有其他两个特性值得注意。

由于你选择的字段存在于两个不同的表中，因此必须指定结果来源的两个表。字段名的顺序就是你获取到的数据的顺序。但是表的顺序无关紧要，只要特定字段出现在那些表中。

因为指定了两个唯一的字段名，所以 SQLite 可以计算出从哪个表中得到它们。如果也想要展示出现在两个表中的 Grade，就必须使用点符号来指定你想要哪个表的 Grade，如下所示：

```
sqlite> SELECT Employee.Grade, Name, Amount
...> FROM Employee, Salary
etc/...
```

特别需要注意的是，SQL 可能会要求这样的限制条件，即使对于 Grade 字段的表的选择真的不重要。因为 WHERE 条件保证对于展示的任意结果行，两个表的 grade 值在任何情况下都是相同的。

这里讨论的 SELECT 的最后一个特性(虽然可以在 SQL 文档中了解到更多关于 SELECT 的信息)是对结果排序的能力。数据库保存数据的顺序通常是为了更加方便地查找东西或按照插入的顺序显示。不管哪一种通常都不是你想要的展示方式！为了处理它，可以在 SELECT 语句中使用 ORDER BY 子句。如下：

```
SELECT columns FROM tables WHERE expression ORDER BY columns;
```

注意，最后的 ORDER BY 子句可以接受多个列。这让你有第一、第二、第三等排序顺序。

可以使用它来获得根据 HireDate 排序的员工名字列表。

```
sqlite> SELECT Name
...> FROM Employee
...> ORDER BY HireDate;
Anne Jones
John Brown
Fred Smith
```


(有趣的是, 尽管 HireDate 并不是 SELECT 中被选择展示的列, 但是 HireDate 可以被完美地接受为一个 ORDER BY 列)。

显然没有更方便的方式了! 唯一值得注意的是, 你并没有使用一个 WHERE 子句。如果使用, 它必须在 ORDER BY 子句之前。因此, 尽管 SQL 并没有要求一个 SELECT 语句的所有部分都出现, 但是它要求那些出现的元素必须以规定的顺序出现。

关于读取数据就到此为止了。现在你会看到如何在适当的位置修改你的数据。

4. 修改数据

你有两种方式修改数据库中的数据。可以修改一个或多个记录的内容, 或更彻底的, 可以删除一个记录或甚至整个表的内容。修改一个已存记录的内容是更加常见的场景, 而可以使用 SQL 的 UPDATE 命令来实现。

基本格式如下:

```
UPDATE table SET column = value WHERE condition;
```

可以通过把 Employee 数据库中一个领班的薪水修改为\$70000 来试试:

```
sqlite> UPDATE Salary
...> SET Amount = 70000
...> WHERE Grade = 'Foreman';
```

要小心写对 WHERE 子句。如果没有指定, 表中的每一行都会被修改, 而这通常不是一个好主意。同样的, 如果 WHERE 子句不是足够明确, 你最终会得到比想要的更多的行。一种用来检查书写是否正确的方式是使用 WHERE 子句来执行一个 SELECT, 并检查是否只找到了你想要的行。如果一切顺利, 可以在你的 UPDATE 语句中重复 SELECT 的 WHERE 子句。

除了仅仅修改一个或多个给定行的某些字段, 你可能需要对你的数据库表做更加激烈的改变: 彻底从表中删除一个或多个行。可以使用 SQL 的 DELETE FROM 命令来实现它。它的基本形式如下:

```
DELETE FROM Table WHERE condition
```

所以, 如果想从 Employee 表中删除 Anne Jones, 可以这样做:

```
sqlite> DELETE FROM Employee
...> WHERE Name = 'Anne Jones';
```

如果有多于一行匹配 WHERE 条件, 那么所有的匹配行都会被删除。SQL 总是会对所有符合特定 WHERE 条件的行进行操作。在这方面, SQL 与比如在 Python 程序中使用一个正则表达式对字符串执行子串替换不太相同(默认行为是只修改第一个找到的地方, 除非你另外明确要求)。

你可能想要对数据库做甚至更加激烈的改变, 是删除不仅表中的所有行还有整个表本

身。这是使用 SQL 的 DROP 命令实现的。

显然，一定要谨慎使用像 DELETE 和 DROP 这样的毁灭性命令。

3.3.3 跨表链接数据

之前已经在 SELECT 部分提到过链接不同表数据的可能性。然而，这是关系型数据库理论的一个基本部分。你会在这更深入地学习它。表之间的链接代表着数据实体之间的关系。这就是关系型数据库，比如 SQLite，名字的由来。数据库维持的不仅仅是实体的原始数据还有关于关系的信息。

有关关系的信息以数据库约束的形式被存储起来。它被应用在使用 CREATE 语句定义数据库结构时。在介绍如何使用约束来塑造关系模型之前，首先需要深入了解 SQLite 中可用的约束类型。

1. 深入理解数据约束

通常，会基于 CREATE 语句中字段与字段的基础上表达约束。这意味着可以扩展基本的 CREATE 定义，从：

```
CREATE Tablename (Column, Column,...);
```

到：

```
CREATE Tablename (
  ColumnName Type Constraints,
  ColumnName Type Constraints,
  ...);
```

最常用的约束有：

- NOT NULL
- PRIMARY KEY [AUTOINCREMENT]
- UNIQUE
- DEFAULT value

NOT NULL 是不言自明的。它表示值必须存在而且不能为 NULL。NULL 值是指没有指定值，就像 Python 中的 None。如果没有为一个有 NOT NULL 约束的字段提供合适的值，数据插入绝对会失败。失败的很可能不仅是那个字段，还有整个行(或者更严重的话，违反约束可以导致一个非常大的数据库更新事务整体失败，这可能涉及数百或数千行)。

PRIMARY KEY 告诉 SQLite 使用此列作为查找的主键(实际上这意味着为了更快的搜索而优化它)。可选的 AUTOINCREMENT 关键字意味着一个 INTEGER 类型值在每个 INSERT 被自动赋值，而值会每次自动加一。这为程序员节省了大量单独计数的工作。注意，AUTOINCREMENT 关键字不是被正常使用的，而是隐含在 INTEGER 和 PRIMARY KEY 的类型或约束组合中。这个 SQLite 文档中并不明显的怪事已经让足够多的人犯错误了。它已经出现在 SQLite 常见问题页的顶端。可以在这找到：<http://sqlite.org/faq.html>。

UNIQUE 约束意味着字段值在指定列必须唯一。如果试图向带有 UNIQUE 约束的列中插入一个重复值，会产生一个错误结果并且行也不会被插入。UNIQUE 经常被用于非 INTEGER 类型的 PRIMARY KEY 列。

DEFAULT 通常伴有一个值。如果用户没有显式提供值，SQLite 就为这个字段插入默认值。

以下是一个小示例，演示了这些约束，包括 DEFAULT 的使用：

```
sqlite> CREATE table test
...> (Id INTEGER PRIMARY KEY,
...> Name NOT NULL,
...> Value INTEGER DEFAULT 42);
sqlite> INSERT INTO test (Name, Value) VALUES ('Alan', 24);
sqlite> INSERT INTO test (Name) VALUES ('Heather');
sqlite> INSERT INTO test (Name, Value) VALUES ('Laura', NULL);
sqlite> SELECT * FROM test;
1|Alan|24
2|Heather|42
3|Laura|
```

首先要注意的是，尽管 INSERT 语句都没有 Id 值，但是在 SELECT 的输出中有 Id 值。这是因为通过指定 Id 为 INTERGER PRIMARY KEY，SQLite 会自动生成它。也要注意 Heather 实体的默认值是如何设置的。同时，注意 Linda 的值是不存在的或为 NULL。NOT NULL 和 DEFAULT 之间有一个重要区别。前者不允许 NULL 值，不管是默认还是显式的设置。DEFAULT 约束防止有未指明的 NULL 值，但是并不阻止故意地创建 NULL 值。

也可以对表本身应用约束，比如，如何处理类似在一个 UNIQUE 列中重复的数据冲突。例如，一个表约束可以指定冲突发生时整个数据库查询都会被取消，或者它可以指定只取消冲突发生行的改变。本章之后不会再讨论表约束。可以在文档中查看详细信息。

2. 回顾 SQLite 字段类型

如之前所述，另一种可以应用的约束是指定列类型。这非常像编程语言中数据类型的概念。SQLite 中可用的类型如下：

- TEXT
- INTEGER
- REAL
- NUMERIC
- BLOB
- NULL

这些类型都是不言而喻的，除了 NUMERIC。它可以存储浮点数、整数和 BLOB。BLOB 代表二进制大对象，常被用于媒体数据，比如图片。NULL 不是一个真实的类型，它只是表明没有指定一个明确的类型。大多数数据库有更加广泛的类型集，包括一个至关重要的

DATE 类型。然而正如你将要看到的，SQLite 对于类型有某种程度上非常规的方法来让这些细节没那么重要。



注意：SQL 标准是由委员会定义的。委员会包括许多现有产品的数据库供应商。SQL 标准有非常广泛的类型列表。SQLite 试图将几个不同名字的别名赋值为同样的基础类型。通过该操作来解决这个问题。因此，原生的 TEXT 类型也可以被表达为 STRING 或 VARCHAR，因为这些是其他供应商使用的术语。这个办法让 SQL 代码从其他数据库移植到 SQLite 变得尽量轻松。

大多数数据库严格地应用指定的类型。然而，SQLite 采用了一种更加动态的方案。指定的类型更像一种提示，而任意类型的数据都可以被存储在表中。当一个不同类型的数据被加载进一个字段时，SQLite 试图使用声明的类型来转换数据。如果它不能被转换，它会以它原始的形式被存储起来。因此，如果一个字段被声明为 INTEGER，但是 TEXT 值'123'被传入，SQLite 会把字符串'123'转换成数字 123。然而，如果 TEXT 值'Freddy'被传入，转换就会失败，所以 SQLite 只是在字段中存储字符串'Freddy'！如果没有意识到这个缺点，这可能引起一些奇怪的行为。大多数数据库将类型声明作为一个严格的约束。传入非法值时会失败。

3. 使用约束对关系建模

在介绍完各种可用约束以及如何在你的数据库中使用它们之后，是时候回到关系建模的主题。所以约束如何能帮助你进行数据建模，尤其是关系建模？再看一下两表数据库，如表 3-5 和表 3-6 所示。

首先看一下 Employee 表，你看到 EmpID 值应该是 INTEGER 类型并且有一个 PRIMARY KEY 约束。其他列除了 ManagerID 都应当是 NOT NULL。ManagerID 也应该是 INTEGER 类型。

表 3-5 Employee 数据库表

EMPID	NAME	HIREDATE	GRADE	MANAGERID
1020304	John Brown	20030623	Foreman	1020311
1020305	Fred Smith	20040302	Laborer	1020304
1020307	Anne Jones	19991125	Laborer	1020304

表 3-6 Salary 数据库表

SALARYID	GRADE	AMOUNT
000010	Foreman	60000
000011	Laborer	35000

在 Salary 表中,你又一次看到 SalaryID 应该是一个 INTEGER 并带有 PRIMARY KEY 约束。Amount 列也应该是一个 INTEGER,而你应该应用一个 20000 的 DEFAULT 值。最后,Grade 列应该被约束为 UNIQUE,因为对于每一个 grade 你不想要多于一个的薪水值!(实际上,这是一个坏主意,因为通常薪水会随着服务年限等变化,但是现在你忽略这些细节。事实上在现实生活中,你或许应该称之为 Grade 表而不是 Salary 表)。

修改后的 SQL 看起来如下所示:

```
sqlite> CREATE TABLE Employee (
...> EmpID INTEGER PRIMARY KEY,
...> Name NOT NULL,
...> HireDate NOT NULL,
...> Grade NOT NULL,
...> ManagerID INTEGER
...> );

sqlite> CREATE TABLE Salary (
...> SalaryID INTEGER PRIMARY KEY,
...> Grade UNIQUE,
...> Amount INTEGER DEFAULT 20000
...> );
```

现在可以试试这些约束。试图输入破坏这些约束的数据并看看会发生什么。希望你看得到一个错误消息!

这里需要指出的一点是,你之前使用的 INSERT 语句已经不能胜任了。之前,你为 ID 字段插入了自己的值,但是现在这些都自动生成了。所以可以(也应该)把它们从插入的数据中删除。但这引起了另一个难点。如果不知道管理者的 EmpID,如何填充 ManagerID 字段呢?答案是可以使用一个嵌套的 SELECT 语句。在这个示例中,可以通过两步完成它。首先使用 NULL 字段,然后在创建所有行之后使用一个 UPDATE 语句。

为了避免大量的重复输入,可以将所有的命令放入两个文件中,保存表创建命令的 employee.sql 和保存 INSERT 语句的 load_employee.sql。这与创建一个以.py 结束的 Python 脚本文件来节省在>>>提示符中的输入量的想法是一致的。

employee.sql 文件如下(此文件在 Chapter3.zip 下载文件的 SQL 目录中):

```
DROP TABLE Employee;
CREATE TABLE Employee (
EmpID INTEGER PRIMARY KEY,
Name NOT NULL,
HireDate NOT NULL,
Grade NOT NULL,
ManagerID INTEGER
);
```

```
DROP TABLE Salary;
CREATE TABLE Salary (
SalaryID INTEGER PRIMARY KEY,
```

```
Grade UNIQUE,
Amount INTEGER DEFAULT 10000
);
```

注意，你在创建表之前删除了它们。如前所述，DROP TABLE 命令删除了表和其中的所有数据。这保证在你创建新表之前，数据库处于一种完全干净的状态(你会在首次运行这个脚本时得到一些错误报告，因为不存在可以 DROP 的表，但是可以忽略它们。后续的执行就应该没有错误了)。

load_employee.sql 脚本如下(此文件在 Chapter3.zip 下载文件的 SQL 目录中)：

```
INSERT INTO Employee (Name, HireDate, Grade, ManagerID)
VALUES ('John Brown', '20030623', 'Foreman', NULL);
INSERT INTO Employee (Name, HireDate, Grade, ManagerID)
VALUES ('Fred Smith', '20040302', 'Labourer', NULL);
INSERT INTO Employee (Name, HireDate, Grade, ManagerID)
VALUES ('Anne Jones', '19991125', 'Labourer', NULL);

UPDATE Employee
SET ManagerID = (SELECT EmpID
                 From Employee
                 WHERE Name = 'John Brown')
WHERE Name IN ('Fred Smith', 'Anne Jones');

INSERT INTO Salary (Grade, Amount)
VALUES ('Foreman', '60000');
INSERT INTO Salary (Grade, Amount)
VALUES ('Labourer', '35000');
```

注意在 UPDATE 命令内部使用的嵌套 SELECT 语句。也要注意通过 SQL IN 操作符使用一个 UPDATE 同时修改了员工表中的两行。IN 操作符工作起来就像 Python 的 in 关键字，用来测试集合成员。通过扩展被测试的名称集，可以轻松地为同一个管理者添加更多员工。

这是你在填充使用约束的数据库时经常遇见的问题。需要仔细设计语句顺序，确保对于含有指向另一个表的引用的每一行，你已经提供了引用的数据！这有点像从一棵树的叶子开始，向下工作到树干。永远先创建或插入没有引用的数据，然后再创建或插入指向那些数据的数据等等。如果在初始化创建之后你正在添加数据，需要使用查询来检查需要的数据是否已经存在，如果它不存在再添加。在这点上，像 Python 一样的脚本语言就变得非常宝贵了。

最后，可以按如下方式从 SQLite 提示符运行这些：

```
sqlite> .read employee.sql
sqlite> .read load_employee.sql
```

确保你已经整理好路径问题。或者从.sql 脚本被保存的地方启动 sqlite3(就像之前一样)，或者提供脚本的完整路径。

现在，试着查询并检查所有东西都正常工作：


```
sqlite> SELECT Name
...> FROM Employee
...> WHERE Grade IN
...> (SELECT Grade FROM Salary WHERE amount >50000)
...> ;
John Brown
```


看起来一切正常。John Brown 是唯一一个薪水超过\$50000的员工。注意你联合使用了一个 IN 条件和另一个嵌套的 SELECT 语句。这是类似你之前执行的跨表 join 的查询的变种。两种技术都能工作，但是通常 join 方法会更快。

尽管这改善了原始无约束定义并且确保 ManagerID 是整数，但是它并不确保整数是有效的 EmpID 关键字。需要为此嵌套使用 SELECT 语句。然而，SQLite 提供了一个额外约束。它可以帮助你确保数据是一致的，这就是 REFERENCES 约束。它告诉 SQLite 一个给定的字段引用了数据库中某处的另一个指定字段。可以通过像这样修改 CREATE 语句把 REFERENCES 约束应用到 ManagerID 字段：

```
CREATE TABLE Employee (
...
ManagerID INTEGER REFERENCES Employee(EmpID)
);
```

你看到 REFERENCES 约束指定了表和内部的关键字字段。在写入时，SQLite 实际上默认不强制检查这个约束(尽管在代码中包含你的意图是值得的)。然而，可以使用编译指示(pragma)语句打开检查。编译指示是一个特殊的命令，用来控制解释器的工作方式。它看起来如下所示：

```
PRAGMA Foreign_Keys=True;
```

 **注意：**术语外键表示它是一个数据库中不同行的关键字。这可以使同一个表的关键字，就像 ManagerID 一样。也可以是一个完全不同表的引用。不管哪种情况，引自不同行的关键字因此被描述为外键。

如果现在修改 employee.sql 文件来添加编译指示并修改创建语句，它看起来应该如下所示：

```
PRAGMA Foreign_Keys=True;

DROP TABLE Employee;
CREATE TABLE Employee (
EmpID INTEGER PRIMARY KEY,
Name NOT NULL,
HireDate NOT NULL,
Grade NOT NULL,
```

```
ManagerID INTEGER REFERENCES Employee(EmpID)
);
```

```
DROP TABLE Salary;
CREATE TABLE Salary (
    SalaryID INTEGER PRIMARY KEY,
    Grade UNIQUE,
    Amount INTEGER DEFAULT 10000
);
```

在运行脚本并使用 load_employee.sql 脚本重载数据之后，可以通过试图插入一个新的但 ManagerID 还在表中的员工检查效果。如下：

```
sqlite> .read employee.sql
sqlite> .read load_employee.sql
sqlite> insert into Employee (Name,HireDate,Grade,ManagerID)
...> values('Fred', '20140602','Laborer',999);
Error: FOREIGN KEY constraint failed
sqlite>
```

这对于保持数据一致性很有帮助。现在不可能创建一个无效的工作者-管理者关系了(尽管你仍然可以使用 NULL 值来表示不存在管理者关系)。

3.3.4 多对多关系

你还没有遇到的一种情形是：两个表通过多对多关系链接起来。也就是说，第一个表中的一行可以被链接到第二表中的多个行，而第二个表中的一行同时也可以被链接到第一个表中的多行。

看看这个示例。想象创建一个数据库来支持一个图书出版公司。它需要作者列表和书列表。每个作者可能写一本或多本书。每本书可以有一个或多个作者。你如何在数据库中表示它呢？解决办法是把书和作者作为一个表格独立的表示。这样的表格通常被称为一个交叉表(intersection table)或映射表(mapping table)。这个表的每一行表示一个书-作者关系。注意每本书可能有多个书-作者关系，但是每个关系只有一本书和一个作者。所以你已经把多对多关系转换成两个一对多关系。而你已经知道如何使用 IDs 构建一对多关系。代码如下(可以在 .zip 文件 SQL 目录的 books.sql 文件中找到代码)：

```
PRAGMA Foreign_Keys=True;
```

```
drop table author;
create table author (
    ID Integer PRIMARY KEY,
    Name Text NOT NULL
);
```

```
drop table book;
create table book (
    ID Integer PRIMARY KEY,
```

```
Title Text NOT NULL
);

drop table book_author;
create table book_author (
bookID Integer NOT NULL REFERENCES book(ID),
authorID Integer NOT NULL REFERENCES author(ID)
);

insert into author (Name) values ('Jane Austin');
insert into author (Name) values ('Grady Booch');
insert into author (Name) values ('Ivar Jacobson');
insert into author (Name) values ('James Rumbaugh');

insert into book (Title) values ('Pride & Prejudice');
insert into book (Title) values ('Emma');
insert into book (Title) values ('Sense & Sensibility');
insert into book (Title) values ('Object Oriented Design with Applications');
insert into book (Title) values ('The UML User Guide');

insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'Pride & Prejudice'),
(select ID from author where Name = 'Jane Austin')
);

insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'Emma'),
(select ID from author where Name = 'Jane Austin')
);

insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'Sense & Sensibility'),
(select ID from author where Name = 'Jane Austin')
);

insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'Object Oriented Design with Applications'),
(select ID from author where Name = 'Grady Booch')
);

insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'The UML User Guide'),
(select ID from author where Name = 'Grady Booch')
);

insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'The UML User Guide'),
(select ID from author where Name = 'Ivar Jacobson')
);
```



```
insert into book_author (BookID,AuthorID) values (
(select ID from book where title = 'The UML User Guide'),
(select ID from author where Name = 'James Rumbaugh')
);
```

如果查看插入到表中的值，会看到 Jane Austin 名下有三本书，而 *The UML User Guide* 这本书有三个作者。

如果把它加载进 SQLite 的名为 books.db(或者就是用 Chapter3.zip 文件的 SQL 目录中的 books.db 文件)的数据库中，可以试一些查询来看看它是如何工作的：

```
$ sqlite3 books.db
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .read books.sql
Error: near line 3: no such table: author
Error: near line 9: no such table: book
Error: near line 15: no such table: book_author

sqlite> .tables
author      book      book_author
```

注意 DROP 语句产生的错误。你在第一次运行脚本时始终会得到那些错误，因为那些表还不存在。现在可以查到 Jane Austin 的哪些书被出版了：

```
sqlite> SELECT title FROM book, book_author
...> WHERE book_author.bookID = book.ID
...> AND book_author.authorID = (
...> SELECT ID from Author
...> WHERE name='Jane Austin');
Pride & Prejudice
Emma
Sense & Sensibility
```

事情变得有点更加麻烦了，但是如果坐下来研究，很快就能理解它。注意，需要在 SELECT 之后在表列表中包含被引用的表 book 和 book_author(第三个表 author 没有被列出来，因为它被列出在自己的嵌套 SELECT 语句中)。现在用相反的方式试试它——看看谁写了 *The UML User Guide* 这本书：

```
sqlite> SELECT name FROM author, book_author
...> WHERE book_author.authorID = author.ID
...> AND book_author.bookID = (
...> SELECT ID FROM book
...> WHERE title='The UML User Guide');
Grady Booch
Ivar Jacobson
James Rumbaugh
```

如果仔细查看，会发现两个查询的结构是相同的，你只是简单交换了表和字段名。

对于这个示例的分析已经足够了。现在返回到你的借出库。看看如何把它从基于文件的存储转换到一个完全的 SQL 数据库。接下来构建一个伴随的 Python 模块。它允许应用编写者忽略 SQL 而只是调用 Python 函数。

3.4 从 LendyDB 迁移到 SQL 数据库

本节使用 SQL 和 Python 代码一起重新创建 LendyDB 数据库。在接触具体细节之前，需要看看 Python 和 SQL 的协作方式。

3.4.1 从 Python 访问 SQL

SQLite 提供了一个包含大量标准函数的应用编程接口或 API。这些函数允许程序员使用交互式 SQL 提示符去执行你已经做过的相同操作。SQLite API 是用 C 语言编写的，但是提供了其他语言的封装器，包括 Python。Python 对很多其他数据库有类似的接口，而它们都提供了一个函数的标准集和非常相似的功能。这个接口被称作 Python DBAPI。相比于每个数据库有自己的接口，它的存在使数据库之间移植数据变得简单得多。

DBAPI 定义了两个有用的概念对象，它们形成了接口的核心。它们是连接(connections)和游标(cursors)。

1. 使用 SQL 连接

连接是程序代码和数据库的 SQL 引擎之间的通道。名称来源于大多数 SQL 数据库使用的主从式架构。其中，客户端必须通过网络连接到服务器。对于 SQLite，连接实际上是通过 SQLite 库连接到数据文件。用于创建连接的传入参数必定是数据库相关的。例如，许多数据库要求一个用户 ID 和密码，而一些要求 IP 地址和端口。而 SQLite 只需要一个文件名，它看起来如下所示：

```
>>> import sqlite3
>>> db = sqlite3.connect('D:/PythonCode/Chapter3/SQL/lendy.db')
```

一旦连接被建立了，则可以继续创建游标。它是用来发送 SQL 命令并接受它们输出的机制。原则上，每个连接可以有多个游标，但实际上很少需要用到。

2. 使用游标

当从程序内部访问数据库时，一个重要问题是如何访问一个 SELECT 语句可能返回的多行数据而不耗尽内存。答案是使用 SQL 中游标(cursor)。游标是一个 Python 迭代器，所以它可以每次迭代地访问一行。因此，通过把数据选入一个游标并使用游标方法去提取结果，或者作为一个列表(用于小量数据)，或者逐行访问，可以处理大数据容器。你现在可以试试它。

试一试：使用 Python DBAPI

在这个“试一试”中，使用 `sqlite3` 模块创建一个表，用数据填充，然后用 `SELECT` 执行一些查询。然后删除数据，删除表并退出。完成下面的步骤：

(1) 启动 Python 解释器并输入下面的代码：

```
>>> import sqlite3
>>> db = sqlite3.connect(':memory:')
>>> cur = db.cursor()
>>> cur.execute("create table test(id,name)")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> cur.execute(
... "insert into test (id,name) values (1,'Alan')")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> cur.execute(
... "insert into test (id,name) values (2,'Laura')")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> cur.execute(
... "insert into test (id,name) values (3,'Jennifer')")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> cur.execute("Select * FROM test")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> print(cur.fetchall())
[(1, 'Alan'), (2, 'Laura'), (3, 'Jennifer')]
```

(2) 如你所见，游标返回了一个元组列表。这与你在第 2 章的使用逗号分隔值部分的开始非常相似！并且可以简单地在程序中使用这个列表，就像你已经从文件中读取了它。使用数据库只是一种持久化机制。然而，数据库的真正力量在于它可以使用 `SELECT` 执行复杂的查询。使用下面的命令试试一些查询：

```
>>> def findData(cursor, aString):
...     cursor.execute("select * from test where name like ?", (aString,))
...     return cur.fetchall()
...
>>> findData(cur, 'A%')
[(1, 'Alan')]
>>> findData(cur, '%a%')
[(1, 'Alan'), (2, 'Laura')]
```

(3) 最后，尝试一些数据操纵命令：

```
>>> cur.execute("delete from test")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> cur.execute("drop table test")
<sqlite3.Cursor object at 0x7fd28ca0>
>>> cur.close()
>>> db.commit()
>>> db.close()
```


示例说明

首先导入 `sqlite` 模块。下一步是创建一个数据库文件的连接。在这个示例中是 `:memory:`，这表示想要一个存在计算机内存中的临时数据库。然后创建了一个游标对象。它随后被用于所有 SQL 交互。

创建了一个测试表并插入三条记录。然后使用 `Select *` 从表中提取所有数据。使用游标的 `fetchall()` 方法查看元组的列表输出(其他选项包括在循环内部使用 `fetchone()` 或用 `fetchmany()` 返回一批结果)。

下一步涉及定义一个 Python 函数。它使用一个字符串来定义感兴趣的记录。这演示了如何创建 Python 函数向用户隐藏 SQL 的细节。之后会使用这个技术为借出库应用创建一个 API。

明白函数如何工作之后，删除了记录，删除了表，并关闭了游标。然后执行了连接 `commit()` 方法。它会永久地应用产生的改变。最后它自己关闭了数据库连接。

`commit` 的使用稍微有些麻烦。DBAPI 有时会调用 `commit`，而有些时候需要显式地做这件事。一般情况下，应该总是在关闭数据库连接前调用 `commit`。在这个示例中，这不会产生实质上的不同，因为数据库只存在于内存中并且会在你关闭连接时消失。如果在使用一个基于文件的数据库，则需要 `commit()` 来确保所做改变在下次打开数据库时可见。

3.4.2 创建 LendyDB SQL 数据库

数据库设计与它之前的设计没有明显不同。只是必须把它翻译为 SQL 语法并添加一些约束来改善数据完整性。

通常初始化数据库设置最简单的办法就是像你在之前部分那样使用原生的 SQL 命令。尽管全部从 Python 做这件事也是可以的，但偶尔另一个工具会更加适合手头的任务。



注意: 如果没有安装 SQLite 解释器，也可以使用游标对象的 `executescript()` 方法执行一个 SQL 脚本。这个方法接受一个 SQL 脚本作为参数。脚本可以从一个文件或标准 Python 字符串读入。字符串可以是多行三引号字符串。注意，需要使用分号结束 SQL 语句，就像你在 `sqlite3` 解释器做的那样。

代码如下(它在 .zip 文件的 SQL 文件夹的 `lendydb.sql` 文件中):

```
PRAGMA Foreign_Keys=True;
```

```
drop table loan;
drop table item;
drop table member;
```

```
create table member (
ID INTEGER PRIMARY KEY,
```

```
Name TEXT NOT NULL,
Email TEXT);
```

```
create table item (
ID INTEGER PRIMARY KEY,
Name TEXT NOT NULL,
Description TEXT NOT NULL,
OwnerID INTEGER NOT NULL REFERENCES member(ID),
Price NUMERIC,
Condition TEXT,
DateRegistered TEXT);
```

```
create table loan (
ID INTEGER PRIMARY KEY,
ItemID INTEGER NOT NULL REFERENCES item(ID),
BorrowerID INTEGER NOT NULL REFERENCES member(ID),
DateBorrowed TEXT NOT NULL,
DateReturned TEXT);
```

需要在脚本一开始删除所有表，并且顺序很重要，因为可能引用的约束会失败并产生错误结果。也要注意，你为物品价格使用了 NUMERIC 类型，因为这满足整数和浮点数。它也意味着你不需要担心那些讨厌的美元符号。你在之前的数据示例中必须除去它。除了各种各样的关键字字段，其他字段都被声明为 TEXT。

既然数据库已经准备好了，现在可以插入测试数据了。

3.4.3 插入测试数据

现在数据库设计包含一些引用约束，所以需要考虑好填充数据的顺序。成员数据没有引用，所以它可以被第一个填充。物品数据只引用了成员，所以它是下一个。最后，借出数据引用了物品和成员，所以它必须是最后一个。

对于数据插入，你实际上在反复用不同数据值重复同样的 INSERT 操作。这一次，Python 是更好的解决方法，因为可以编写 SQL 一次并多次执行它，或者用游标对象的 executemany() 方法，或者在 Python for 循环内部调用 execute() 方法。代码如下(代码在 lendydata-sql.py 文件中)：

```
import sqlite3
```

```
members = [
    ['Fred', 'fred@lendylib.org'],
    ['Mike', 'mike@gmail.com'],
    ['Joe', 'joe@joesmail.com'],
    ['Rob', 'rjb@somcorp.com'],
    ['Anne', 'annie@bigbiz.com'],
]
member_sql = '''insert into member (Name, Email) values (?, ?)'''

items = [
```

```

['Lawnmower','Tool',    0, 150,'Excellent','2012-01-05'],
['Lawnmower','Tool',    0, 370,'Fair',      '2012-04-01'],
['Bike',      'Vehicle', 0, 200,'Good',      '2013-03-22'],
['Drill',     'Tool',    0, 100,'Good',      '2013-10-28'],
['Scarifier','Tool',    0, 200,'Average',    '2013-09-14'],
['Sprinkler','Tool',    0, 80,'Good',        '2014-01-06']
]
item_sql = ''
insert into item
(Name, Description, ownerID, Price, Condition, DateRegistered)
values (?, ?, ?, ?, ?, date(?))''
set_owner_sql = ''
update item
set OwnerID = (SELECT ID from member where name = ?)
where item.id = ?
'''

loans = [
    [1,3,'2012-01-04','2012-04-26'],
    [2,5,'2012-09-05','2013-01-05'],
    [3,4,'2013-07-03','2013-07-22'],
    [4,1,'2013-11-19','2013-11-29'],
    [5,2,'2013-12-05',None]
]
loan_sql = ''
insert into loan
(itemID, BorrowerID, DateBorrowed, DateReturned )
values (?, ?, date(?), date(?))''

db = sqlite3.connect('lendy.db')
cur = db.cursor()

cur.executemany(member_sql, members)
cur.executemany(item_sql, items)
cur.executemany(loan_sql, loans)

owners = ('Fred','Mike','Joe','Rob','Anne','Fred')
for item in cur.execute("select id from item").fetchall():
    itemID = item[0]
    cur.execute(set_owner_sql, (owners[itemID-1], itemID))

cur.close()
db.commit()
db.close()

```

这个脚本中有重要的几点值得注意。第一点是日期格式。尽管 SQLite 没有 Date 数据类型,但是它有一些可以用来创建标准日期字符串和值的函数。然后 SQLite 把它们保存为文本(或者在一些情况下存储为浮点数)。这里使用的 `date()` 函数要求日期字符串是所示的格式,而一个非法日期字符串会被存储为 NULL。因此,通过确保只有有效的、格式一致的

日期被存储, 使用 `date()` 改善了数据质量。



注意: 在 SQLite 中, 使用日期函数处理日期字段是非常不常见的, 但是它适合 SQLite 的动态类型机制。

其他日期函数包括你在第 2 章使用时间模块部分见到的 `strftime()` 函数的一种版本。SQLite 版本添加了一些额外的特性。

关于 SQLite 的时间结构, 有一点要注意的是它永远使用 UTC(又称 GMT), 因为这可以避免关于时区和夏令时的任何复杂问题。为了用本地时间格式展示数据, 需要使用 Python 的时间函数转换 SQLite 返回的时间。SQLite 文档中有一页描述了各种各样的日期函数如何能被用在几种常见的场景中 (http://www.sqlite.org/lang_datefunc.html)。

物品的 `OwnerID` 字段被指定为 NOT NULL, 所以用一个虚拟值(0)填充它。这之后在脚本的 UPDATE 代码中被重写了。

本可以使用一个 for 循环来处理 INSERT 语句, 但使用了 `executemany()` 方法。这个方法接受语句和一个序列(或迭代器或生成器), 并重复地应用语句直到迭代结束。

通过使用一个问号作为占位符, 变量值被插入查询字符串中。这和使用 `[]` 作为占位符的字符串格式化方法非常相似。



注意: 你可能好奇为什么字符串格式化没有用来向 SQL 语句中插入值。这是因为字符串格式化易受一种名为注入式攻击的安全攻击侵害。注入式攻击中, 有害代码可以被注入到 SQL 语句中。使用 SQLite 的 `execute()` 机制作为格式化工具消除了这种风险, 因为它会检查有害值。

此时, 已经创建了数据库并填充了一些测试数据。下一步是通过一个应用编程接口(API)让程序可以访问数据。

3.4.4 创建一个 LendyDB API

在为数据库定义一个 API 时, 通常首先要考虑基础实体和提供用于创建、读、更新和删除项的函数(这通常被称为操作首字符命名的 CRUD 接口)。诱惑是提供这些函数作为一层对应的 SQL 命令的简单包装。然而, 对于应用程序员来说, 只是得到一个成员的 ID 不是特别有用。例如, 在大多数情况下获得成员的名字要更好。否则, 程序员必须执行多个成员数据库的读操作为用户生成一个有用的展示。另一方面, 有时 ID 是最好的选择, 因为程序员可能想要获得成员的更多具体信息。构建一个好的 API 在于能够解决这些矛盾,

在保持对数据的完整访问的同时让应用程序员很高效。

尽管 CRUD 操作是一个很好的基础，但通常对于应用程序员来说，如果一些高级别的操作在实体间可用的话会更有用。为此，需要考虑应用中会如何使用数据。设计者想要问什么样的问题？如果你也是应用程序员，这会相对简单，但是如果你在为一个更大的项目提供数据库部分，这就更加困难了。

对于借出库这个示例，你仅专注于物品和成员的 CRUD 接口。原理应该很明显，而你会在本章最后的练习 3.3 中有机会扩展 API 来覆盖借出表(物品和成员的代码在.zip 文件 SQL 文件夹内的 lendydata.py 文件中)。

```
'''
Lending library database API

Provides a CRUD interface to item and member entities
and init and close functions for database control.
'''

import sqlite3 as sql

db=None
cursor = None

##### CRUD functions for items #####

def insert_item(Name, Description, OwnerID, Price, Condition):
    query = '''
    insert into item
    (Name, Description, OwnerID, Price, Condition, DateRegistered)
    values (?,?,?,?,?, date('now'))'''
    cursor.execute(query, (Name,Description,OwnerID,Price,Condition))

def get_items():
    query = '''
    select ID, Name, Description, OwnerID, Price, Condition, DateRegistered
    from item'''
    return cursor.execute(query).fetchall()

def get_item_details(id):
    query = '''
    select name, description, OwnerID, Price, Condition, DateRegistered
    from item
    where id = ?'''
    return cursor.execute(query, (id,)).fetchall()[0]

def get_item_name(id):
    return get_item_details(id)[0]

def update_item(id, Name=None, Description=None,
                OwnerID=None, Price=None, Condition=None):
```

```

query = '''
    update item
    set Name=?, Description=?, OwnerID=?, Price=?, Condition=?
    where id=?'''
data = get_item_details(id)
if not Name: Name = data[0]
if not Description: Description = data[1]
if not OwnerID: OwnerID = data[2]
if not Price: Price = data[3]
if not Condition: Condition = data[4]

cursor.execute(query, (Name,Description,OwnerID,Price,Condition,id))

def delete_item(id):
    query = '''
    delete from item
    where id = ?'''
    cursor.execute(query, (id,))

##### CRUD functions for members #####

def insert_member(name, email):
    query = '''
    insert into member (name, email)
    values (?, ?)'''
    cursor.execute(query, (name,email))

def get_members():
    query = '''
    select id, name, email
    from member'''
    return cursor.execute(query).fetchall()

def get_member_details(id):
    query = '''
    select name, email
    from member
    where id = ?'''
    return cursor.execute(query, (id,)).fetchall()[0]

def get_member_name(id):
    return get_member_details(id)[0]

def update_member(id, Name=None, Email=None):
    query = '''
    update member
    set name=?, email=?
    where id = ?'''
    data = get_member_details(id)
    if not Name: Name = data[0]

```



```

if not Email: Email = data[1]
cursor.execute(query, (Name, Email, id))

def delete_member(id):
    query = '''
delete from member
where id = ?'''
    cursor.execute(query, (id,))

##### Database init and close #####

def initDB(filename = None):
    global db, cursor
    if not filename:
        filename = 'lendy.db'
    try:
        db = sql.connect(filename)
        cursor = db.cursor()
    except:
        print("Error connecting to", filename)
        cursor = None
        raise

def closeDB():
    try:
        cursor.close()
        db.commit()
        db.close()
    except:
        print("problem closing database...")
        raise

if __name__ == "__main__":
    initDB() # use default file
    print("Members:\n", get_members())
    print("Items:\n", get_items())

```

在这个模块中，为数据库连接和游标创建了两个全局变量。initDB()函数初始化了这些变量，并且同一个游标被用在模块中的每一个查询函数中。接下来在使用完模块后，closeDB()函数关闭了这些对象。这个方法意味着必须调用 initDB() /closeDB()函数来初始化和关闭数据库。这给用户增加了一点麻烦，但是它意味着单独的查询会更加简单，因为不需要在每次调用查询时都管理数据库和游标对象。也要注意，在 initDB()函数中设置了 Foreign_Keys 编译指示来确保对所有事务都会检查引用完整性。

查询函数都遵循同样的模式。一个 SQL 查询字符串被创建，使用三引号允许多行布局，然后这个字符串被传到 cursor.execute()方法中。取到的值在合适的地方被返回。自始至终一直使用 cursor.execute()的参数替换机制。

两个更新方法有一点新花样，它们都有默认的输入参数。这意味着用户可以只提供那

些他们要改变的字段。其他字段的填充会基于使用适合的获得细节函数获取到的现有值。

两个获得名称函数可以方便用户轻松地将从获得细节查询中返回的数据库标识符映射到有意义的名称。一般来说，应用程序员应该在向用户展示结果前使用这些函数。

模块最后的 if 测试内部是一个非常基本的测试函数。它只是用来检查连接和游标对象按预期工作。它没有彻底地测试所有的 API 函数，你会在接下来的“试一试”中实现这一点。

试一试：使用 LendyDB API

在这个“试一试”中，会使用 `lendydata.py` 模块来插入、读取、更新和从之前创建的 `lendy.db` 数据库中删除实体。完成下面的步骤：

- (1) 切换到包含 `lendy.db` 数据库文件的文件夹。
- (2) 启动 Python 解释器并输入下面的代码：

```
>>> import lendydata as ld
>>> ld.initDB()
>>> ld.get_members()
[(1, 'Fred', 'fred@lendylib.org'), (2, 'Mike', 'mike@gmail.com'),
 (3, 'Joe', 'joe@joesmail.com'), (4, 'Rob', 'rjb@somcorp.com'),
 (5, 'Anne', 'annie@bigbiz.com')]
>>> ld.get_items()
[(1, 'Lawnmower', 'Tool', 1, 150, 'Excellent', '2012-01-05'),
 (2, 'Lawnmower', 'Tool', 2, 370, 'Fair', '2012-04-01'),
 (3, 'Bike', 'Vehicle', 3, 200, 'Good', '2013-03-22'),
 (4, 'Drill', 'Tool', 4, 100, 'Good', '2013-10-28'),
 (5, 'Scarifier', 'Tool', 5, 200, 'Average', '2013-09-14'),
 (6, 'Sprinkler', 'Tool', 1, 80, 'Good', '2014-01-06')]
```

- (3) 在确认可以访问数据后，现在通过下面的代码添加一个物品：

```
>>> ld.insert_item('Python Projects', 'Book', 6, 30, 'Excellent')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "lendydata.py", line 20, in insert_item
    cursor.execute(query, (Name, Description, OwnerID, Price, Condition))
sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

(4) 前面的错误证明引用完整性检查正常工作。首先需要添加一个成员，然后添加新物品。输入以下这些代码：

```
>>> ld.insert_member('Alan', 'alan@emailaddress.com')
>>> ld.get_members()
[(1, 'Fred', 'fred@lendylib.org'), (2, 'Mike', 'mike@gmail.com'),
 (3, 'Joe', 'joe@joesmail.com'), (4, 'Rob', 'rjb@somcorp.com'),
 (5, 'Anne', 'annie@bigbiz.com'), (6, 'Alan', 'alan@emailaddress.com')]
>>> ld.insert_item('Python Projects', 'Book', 6, 30, 'Excellent')
>>> ld.get_items()
[(1, 'Lawnmower', 'Tool', 1, 150, 'Excellent', '2012-01-05'),
```

```
(2, 'Lawnmower', 'Tool', 2, 370, 'Fair', '2012-04-01'),
(3, 'Bike', 'Vehicle', 3, 200, 'Good', '2013-03-22'),
(4, 'Drill', 'Tool', 4, 100, 'Good', '2013-10-28'),
(5, 'Scarifier', 'Tool', 5, 200, 'Average', '2013-09-14'),
(6, 'Sprinkler', 'Tool', 1, 80, 'Good', '2014-01-06'),
(7, 'Python Projects', 'Book', 6, 30, 'Excellent', '2014-06-23')]
```

(5) 成功添加了一个物品。现在输入以下代码修改物品和成员：

```
>>> ld.update_item(7, Price=25)
>>> ld.get_item_details(7)
('Python Projects', 'Book', 6, 25, 'Excellent', '2014-06-23')
>>> ld.get_member_name(6)
'Alan'
>>> ld.update_member(6, Name='Alan Gauld')
>>> ld.get_member_details(6)
('Alan Gauld', 'alan@emailaddress.com')
```

(6) 为了删除添加的数据，输入以下代码：

```
>>> ld.delete_member(6)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "lendydata.py", line 97, in delete_member
cursor.execute(query, (id,))
sqlite3.IntegrityError: FOREIGN KEY constraint failed
>>> ld.delete_item(7)
>>> ld.delete_member(6)
```

(7) 输入下面的代码查看 API 如何被用于任意查询：

```
>>> ld.cursor.execute('''
... select * from item
... where OwnerID in (select id from member where name like '%e%')
... ''').fetchall()
[(1, 'Lawnmower', 'Tool', 1, 150, 'Excellent', '2012-01-05'),
 (2, 'Lawnmower', 'Tool', 2, 370, 'Fair', '2012-04-01'),
 (3, 'Bike', 'Vehicle', 3, 200, 'Good', '2013-03-22'),
 (5, 'Scarifier', 'Tool', 5, 200, 'Average', '2013-09-14'),
 (6, 'Sprinkler', 'Tool', 1, 80, 'Good', '2014-01-06')]
>>> ld.cursor.execute('''
... select * from item
... where OwnerID not in (select id from member where name like '%e%')
... ''').fetchall()
[(4, 'Drill', 'Tool', 4, 100, 'Good', '2013-10-28')]
>>> ld.get_member_name(4)
'Rob'
```

(8) 最后，如下所示关闭数据库：

```
>>> ld.closeDB()
```


示例说明

首先引用了 `lendydata.py` 模块并分配了一个更短的别名 `ld`。然后调用 `initDB()` 初始化模块。通过获取成员和物品的列表，你确认连接是有效的。

接下来试图添加一个物品，但 `OwnerID` 值是一个不存在的成员。而数据库 `Foreign_Keys` 约束检查阻止了操作的完成。然后添加了缺少的成员和物品。这一次，物品被成功添加了。

在添加数据后，继续探索更新函数。修改新物品的价格和新成员的名称。也使用了 `get_member_name()` 辅助函数来验证 `OwnerID 6` 是否确实是已经创建的成员。

接下来试图删除新成员，但是又一次引用完整性检查阻止了该操作，因为这个成员被一个物品引用。首先需要删除物品，然后再删除成员。

在练习了高级别的 API 函数后，接下来使用游标对象直接执行任意的 SQL 语句。在本示例中，你找出名字中有 `e` 的成员拥有的所有物品，以及名字中没有 `e` 的成员拥有的所有物品。使用了 `get_member_name()` 辅助函数来验证只有 ID 为 4 的成员是 Rob。

最后，关闭了数据库。确保所有的更改都已被提交，并且连接和游标对象都被正确关闭。

现在已经构建了一个高级别 API 让应用程序员使用，并且已经看到通过让底层游标对象可用，也允许用户发出任意底层 SQL 命令。应用程序员通常很喜欢这种灵活性，尽管如果他们发现必须大量使用 SQL，他们应该请求附加功能性 API。

当然，SQLite 并不是操作大数据量唯一可用的工具，并且 Python 也可以为其他选择提供支持。在下一节中，会看到一些 SQLite 的其他选择以及 Python 如何在这些环境下工作。

3.5 探索其他数据管理选择

你有很多选择来管理大量数据。在这本节中，你会看到传统的主从数据库，较新的名为 NoSQL 的数据库，以及云存储可以改变数据管理的方式。最后，会了解一些强大的数据分析工具。它们可以通过第三方模块从 Python 中访问。

3.5.1 主从数据库

传统的 SQL 数据库在构建方式上不同于 SQLite。在这些数据库中，有一个数据库服务器进程并被多个客户端通过网络访问。Python DBAPI 被设计成可以像 SQLite 一样与这些数据库工作。只有在初始化数据库连接时有很小的改变，而这通常是必要的。偶尔，你会看到 SQL 语法的微小改变，而参数替换符有时也会不同。但是大体上，从一个 SQL 接口切换到另一个是一个相对无痛的过程。

有几个原因会想要采用一个主从数据库或从 SQLite 移植一个。第一个原因是容量。大多数主从数据库可以放大到比 SQLite 更大的数据容量。此外，通过配置，它们的进程可以被分配到数个服务器和磁盘。当多个用户在同时访问数据时，这可以极大地提升性能。第

二个从 SQLite 移植的原因是更大的数据库通常有更加丰富的 SQL 命令集合、查询类型以及更多的数据类型。许多甚至支持面向对象的技术，并且有它们自己的内置编程语言。这可以让你编写存储过程，从而高效地将 API 放入数据库中。通常，一个更加丰富的数据库约束集可以被用来加强数据完整性，远强于 SQLite 执行的简单外键检查。

选择主从服务器的最大缺点是管理数据库的额外复杂度。通常一个有经验的管理员需要创建用户、调整他们的访问权限、调试 SQL 查询引擎性能、数据备份、数据抽取和加载。

流行的主从数据库包括商业产品，比如 SQL Server、Oracle 和 DB2，以及开源项目，比如 MySQL、Postgres 和 Firebird。

3.5.2 NoSQL

由于数据存储需要在大小和多样性上扩展，数个项目已经在探索 SQL 的替代品。这些项目很多都与名为大数据的东西相关。大数据通常涉及从例如社交网站或工厂或医院的传感器等地方收集大量通常无结构的数据。这种数据的一个特性是：对于数据库用户来说大部分数据都没有意义，但是在这些糟粕中有些精华是可以影响销售策略的，或让你意识到即将来临的组件或过程的失败。这样可以提前采取行动。这种数据的性质、数量和快速访问的需求意味着传统 SQL 数据库不能很好地适合任务。

解决方案已经有各种从属于术语 NoSQL 的技术。NoSQL 不意味着没有使用 SQL，其实它代表不仅仅使用 SQL(Not Only SQL)。在这些新数据库上，SQL 仍然可以用于传统的查询。但是可选的查询技术也被使用，尤其是搜索无结构的数据。NoSQL 方法(有典型实现)的一些示例有文档(MongoDB)、键值(Dynamo)、纵栏式(HBase)和图(Allegro)。所有这些在某种程度上不同于多个二维的交叉引用表的传统关系型模型。为了能从这些系统获得最好的效果，需要考虑哪种架构最适合你数据的本质。许多解决方案是开源的，但是也存在商业产品。

尽管 NoSQL 数据库为了更快更灵活地访问更多样的数据提供了可能性，但是它们通常牺牲一些特性，比如数据完整性、事务控制和可用性。尽管相比于传统 SQL 数据库，它们在本书成书时需要更低级别的编程技巧来使用数据，但产品都在快速发展，这一点可能很快就改变了。但部分流行数据库提供了 Python 模块来辅助 Python 编程。

3.5.3 云计算

云计算因为按需计算的出现在近些年很流行。相比于传统的基于数据中心的解决方案，它潜在地提供了更低成本、更大灵活性和更低风险。当然，它也有自己的问题，尤其是敏感数据问题和网络可靠性。在数据库世界，云技术最大的应用就是与之前部分讨论的 NoSQL 解决方案和其他大数据解决方案(比如 Hadoop)的结合。许多云存储提供商基于软件即服务(software-as-a-service, SAAS)的基础提供这些技术。这为那些刚刚进入大数据或 NoSQL 世界的人提供了一个有吸引力的选项。



注意：云服务通常依赖一个前提：只要服务不被打断，数据可以不必通知客户端而在被物理上移动。新数据中心可以联机而数据不被察觉地传输到这些客户端。遗憾的是，许多企业客户坚持想要知道他们的数据被存储在什么地方。在一些国家，这可能要考虑数据保护法规，或者更可能是法规缺失。这些考虑经常被写入一些部门，比如安防、政府和金融的项目合同。如果云提供商不能保证遵守合同条目，则可能需要看看其他的或甚至决定云存储是不合适的。

对于应用程序员来说，云计算的优势是数据被抽离到位于一个不变的网络地址的虚拟数据库中。数据的物理位置可能随时改变，或者可用存储量可能增加或缩小，但是对程序员没有任何影响(这不意味着可以忽略正常错误处理，但是意味着代码可以很大程度上从数据存储的物理设计中解耦)。

云存储和数据库技术的最大提供商之一是在线零售商 Amazon。它提供存储和 API(Amazon 网络服务，或 AWS)、名为 SimpleDB 的专业 NoSQL 数据库以及其他开源商品。其他云供应商正开始提供类似产品。许多供应商提供小量的免费存储和访问来鼓励在投入重大投资之前试一试产品。

Amazon AWS 有可用的 Python 支持，包括一个全面的 Python 和 AWS 接口教程，网址为 <http://boto.readthedocs.org/en/latest/>。其他供应商也有类似的可用的或者将会可用的接口。

3.5.4 使用 RPy 进行数据分析

尽管主从 SQL、NoSQL 和云计算都为处理大量数据或多用户提供解决方案，但你还要考虑其他数据管理问题。有时，处理大量数据要比数据存取更加重要。如果那个过程涉及高度的统计探索或操作，Python 提供了一个基础的 `statistics` 模块(在 3.4 版本的标准库中引入)。如果还不够用，还有 R 编程语言。R 是一个专为统计的大规模数字运算设计的专业语言。像 Python 一样，它已经积累了一个大插件模块库，并且许多统计的研究者使用 R 作为他们选择的平台，并使用 R 发布他们的研究。

好消息是，Python 程序员可以使用一个从 Python 到 R 名为 `rpy2` 的接口。它让你能够使用这种处理能力而不需要成为 R 的行家。了解 R 的基础是非常重要的，尤其是它的数据处理概念。但是也可以应用很多 Python 知识。可以在 Python 包索引中找到它，并通过 `pip` 安装它。

数据管理有很多方面，而本章已经回顾了 Python 如何可以帮助你，从小量到大量，从简单数据持久化到复杂数据分析。一旦你掌控了数据，不管是创建桌面上的还是网上的强大的、用户友好的应用，你都会在一个强有力的位置。这是接下来两章要介绍的主题。

3.6 本章小结

本章介绍了如何存取数据以便应用可以从上次离开的地方开始，或以便在多个不同项目上工作。你看到如何使用索引文件(DBM)、pickles 和 shelve 作为平面文件来实现这个功能。然后使用 SQLite 了解了 SQL 和关系型数据库，并最后回顾了一些可选技术。

平面文件适合存储少量数据或保存程序关闭时的状态信息。当需要多个查询尤其是查询多于一个关键字值时，它们就不是很有用了。DBM 文件的行为就像持久化字典，它只能存储字符串。pickles 将二进制对象转换为字符串。shelve 虽然只能使用字符串作为关键字，但结合了 DBM 和 pickle 之后它的行为像持久化字典。

在关系型数据库管理系统中(RDBMS)，SQL 被用来管理数据。在概念上，关系型数据库由一个或多个二维表组成。每个表代表一个逻辑实体。表间交叉引用就像实体之间的关系模型。SQL 提供命令来创建和管理数据库中的表，以及在那些表中创建、删除、查询和更新数据的操作。

Python 提供了 DBAPI。它是一个用于访问基于 SQL 数据库的标准协议。DBAPI 主要包含两个对象：连接和游标。游标是用来执行 SQL 命令并取回它们的结果的重要对象。结果是一个元组列表。

Python 标准库提供了 SQLite 数据库接口，并且可以下载一个独立的解释器。顾名思义，SQLite 是一个轻量级的 SQL 版本。它把整个数据库存放在一个单独的文件中并支持一个轻量级的 SQL 语言子集。SQLite 可以被用于小到中等规模的项目，并且如果项目规模扩大，它可以相对容易地移植到一个更大的数据库中。DBAPI 库对于大多数主流数据库都可用，并且可以从第三方下载。

通过使用 DBAPI，可以构建一个数据抽象层，对应用程序员隐藏数据库设计细节和 SQL 语言。如果需要的话，它还有助于移植过程。

还有一些其他的数据库技术，并且这是一个发展活跃的区域。有多个 NoSQL 项目在为赢得广泛支持而竞争。这些数据库倾向于有非常具体的应用领域，但没有一个解决方案适用于所有场景。许多数据库技术专注于管理大数据并且非常适合基于云的解决方案。

练习

1. 为了领会 pickle 的功能，试着为数字构建一个简单的名为 `ser_num()` 的序列化函数。它应该接受任何合法整数或浮点数作为参数，并将它转换为一个字节字符串。也应该编写一个函数来执行逆向操作，该函数读取 `ser_num()` 函数产生的字节字符串并将它转换回合适类型的数字(提示：struct 模块对这个练习很有帮助)。

2. 使用 shelve 而非 SQLite 编写员工数据库示例的一个版本。使用样例数据填充 shelf 并编写一个函数列出所有收入高于一个特定数目的员工的名字。

3. 扩展 `lendydata.py` 模块，为借出表提供 CRUD 函数。添加一个额外函数 `get_active_loans()`，列出那些仍然有效的借出(提示：这意味着 `DateReturned` 字段为 NULL)。

4. 研究一下 Python statistics 模块，看看它提供了什么(只存在于 Python 3.4 或更新版本)。

本章所学知识

主 题	关 键 概 念
持久化	在程序执行间存储数据的能力。这样当一个程序重启时，它可以把数据恢复到程序最后被停止的状态
平面文件	包含数据的标准文件。数据可能是文本或二进制格式
DBM	一种文件存储的形式。它使用一组平面文件存储数据和索引文件来存储单个数据记录的位置以快速取回。关键字和数据必须都是字符串格式。在 Python 中，dbm 模块提供了类字典的接口
序列化	将二进制数据转换为字节字符串以方便存储或网络传输的过程
Pickle	一种方便序列化二进制数据的 Python 特定格式。通过使用 Pickle, 大多数 Python 对象可以被序列化。由于它存在一些安全风险，在从不信任源读取数据时必须要小心
Shelve	它是 DBM 和 Pickle 技术的结合。用来提供一个持久的字典。任意二进制数据都可以用基于字符串的关键字存储
关系型数据库	包含一个或多个表的数据库。表的每行代表记录，而列代表记录的字段。字段值可以引用数据库的其他记录，因此代表了实体之间的关系
关系	关系型数据库的每一条记录有一个唯一的主键，而其他记录可以存储另一个记录主键的引用，也因此两个记录和它们的表之间建立了关系。关系可以跨表
约束	可以定义各种规则来确保在数据库维持着数据的完整性。这些规则被称为约束，并规范着一些事情，比如数据类型、是否一个值是必需的，以及是否一个交叉引用必须包含另一个实体的有效关键字
基数	数据库的关系可以表示各种映射类型。每个实体在映射中的数量被称为它的基数。比如，如果一个实体恰好指向另一个实体，这就是一个 1-1 映射。如果多个实体指向另一个实体，这就是一个 1-N 映射。如果很多实体指向很多其他实体，这就是一个多对多映射
结构化查询语言 (SQL)	使用关系型数据库的一种标准化机制。语言包括名为数据定义语言(data definition language, DDL)的数据库结构命令和名为数据操作语言(data manipulation language, DML)的操作和取出数据库的数据命令
DBAPI	用于从 Python 中访问关系型数据库的标准编程协议。DBAPI 的实现在细节上略有不同，但是在 DBAPI 库之间移植代码要比在原生数据库层移植容易得多

(续表)

主 题	关 键 概 念
NoSQL	不只 SQL(Not Only SQL)是一个用来描述多个不遵从传统关系模型的数据库技术的术语。很多这些技术聚焦于管理有多样数据类型的非常大的数据。大部分数据是无结构的，比如文档和社交媒体数据。SQL 并不适合处理这样的无结构数据，因此 NoSQL 技术已经变得越来越重要

创建桌面应用

本章主要内容:

- 如何组织并创建命令行应用
- 如何充实命令行应用
- 如何组织 Tkinter 组织并创建 GUI 应用
- 如何应用 Tk 和以充实 Tkinter 应用
- 如何使用第三方框架扩展 GUI 功能
- 如何本地化和国际化应用

从 www.wrox.com 下载本章代码

可以在 www.wrox.com 网站找到本章涉及的代码。代码可在 www.wrox.com/go/pythonprojects 的 Download Code 选项卡下找到。第4章代码在 Chapter 4 download，名为 Chapter4.py，每个子目录都是按照本章各节的代码文件名命名的。

Python 是一门通用的编程语言，所以它用于多种类型的程序。你已经明白它作为一门脚本语言，可以将应用联系在一起，并且可以管理数据持久化和访问。现在，你将学习如何创建它创建桌面的桌面应用。

桌面应用是个人计算机的基石——包括系统工具，比如文字处理器、工作表以及游戏。它们的功能通常完全在本地进行，不需要网络访问。有时，它们可能本身是面向网络的，比如 Web 浏览器或 Web 式数据库应用。它们的基本特点是大量功能在本地 CPU 上执行。

桌面应用可以拥有很多用户界面(UI)的变体。本章将介绍如何在相同的高层程序逻辑上构建不同的用户界面。这通常可以从一个简单的文本界面开始，然后你与网络上的图形化问题。甚至，你还可以添加一个 Web 用户界面将你的桌面应用扩展成一个网络应用。你会在下一章学习到如何实现这一点。

首先，你会了解应用的基本结构和架构(architecture)，然后，会创建一个简单的命令行

第 4 章

创建桌面应用

本章主要内容:

- 如何组织并创建命令行应用
- 如何充实命令行应用
- 如何使用 Tkinter 组织并创建 GUI 应用
- 如何使用 Tix 和 ttk 充实 Tkinter 应用
- 如何使用第三方框架扩展 GUI 选项
- 如何本地化和国际化应用

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可在 www.wrox.com/go/pythonprojects 的 Download Code 选项卡下找到。第 4 章代码在 Chapter 4 download, 名为 Chapter4.zip, 每个文件都是根据本章提到的代码文件名命名的。

Python 是一门通用的编程语言, 可以被用于多种类型的程序。你已经明白它作为一门脚本语言, 可以将应用联系在一起, 并且可以管理数据持久化和访问。现在, 你将学习如何使用它创建完整的桌面应用。

桌面应用是个人计算的基石, 包括标准工具, 比如文字处理器、工作表以及游戏。它们的功能通常完全在本地进行, 不需要网络访问。有时, 它们可能本身是面向网络的, 比如 Web 浏览器或主从式数据库应用。它的显著特点是大量功能在本地 PC 上执行。

桌面应用可以拥有图形用户界面(GUI)或命令行界面(CLI)。本章将介绍如何在相同的底层程序逻辑上构建不同的用户界面。这意味着可以先从一个简单的文本界面开始, 然后在当前代码上添加一个图形化的前端。甚至, 也可以添加一个 Web 用户界面将你的桌面应用转换成一个网络应用。你会在下一章学习如何实现这一点。

首先, 你会了解应用的基本结构或架构(architecture)。然后, 会创建一个简单的命令行

应用,并使用一些 Python 模块提供更加丰富的用户体验和扩展应用。接下来,会使用 Python 的标准 GUI 工具箱 Tkinter 来构建一个 GUI 前端。这个工具箱集成了所有标准的 GUI 特性,比如控制、菜单和对话框。你也会集成一些额外的 GUI 特性并使用更多的 Python 模块来增强界面的外观。接下来,你会看到其他提供更强大功能的 GUI 框架。最后,将介绍如何支持本地配置和多语言。

4.1 组织应用程序

构建一个高效、可扩展应用的关键是应用分层架构。最常用的方法是把应用分为三层:用户界面层、核心逻辑层(也称为业务逻辑层)和数据层。当应用大量使用网络时,也可能会有网络层。




注意: 这种多层架构还有一个更加正式的版本,它被称为主从式计算。主从模式会维护一个严格的请求/应答操作的层级。每一层都是层下面的客户端,并且负责生成用于接受应答的请求。核心逻辑层既作为用户界面层的服务器又作为数据层的客户端。真正的主从设计并不在本书的讨论范畴内,但是多层方法演示了如何集成多个相同的概念。

用户界面应该为用户展示应用逻辑,但并不实现逻辑。它的任务是尽可能简单地展示应用的功能,并且尽可能清晰地展示结果或输出。用户界面会控制在给定时间点上哪些功能是可用的。例如,在没有任何文档被打开时,不可能关闭一个文档。如果使用面向对象的程序(Object-Oriented Program, OOP),对象通常会表示类似菜单、按钮和窗口的东西。用户界面通过调用逻辑层提供的函数或方法访问核心逻辑层。

核心逻辑层包含所有数据的算法和状态管理。在该层,可以编写代码来改变数据值、创建新实体、打开和关闭文件等。它的目的是提供一组用户界面可以访问的函数或服务。为了提高效率,核心逻辑函数不应打印结果,但是应该把它们作为值(也就是字符串、数值、列表、对象等)返回。用户界面可以在合适的地方将这些值以适当的格式展现出来。核心逻辑仅显示信息。它本身不关心信息的显示格式。正是因为逻辑和显示分开,可以在相同核心逻辑上创建不同的用户界面。核心逻辑层会操作数据层提供的数据。如果使用 OOP,对象会表示问题的概念实体,比如银行账户、人、网络信息和地点等。


数据层用于管理数据。它把数据存储在不安全的地方并在需要时取回数据。它不应该包含复杂算法或特定于应用的逻辑。它仅仅把原始数据传递到核心逻辑层来处理。数据层可能包含一些基本的数据完整性处理来确保数据的一致性。它可能也会集成一些安全功能,比如密码控制或加密。它应该通过一组对象、函数或服务来显示数据。如果使用 OOP,你的对象通常用来表示查询、表、数据连接等。理想情况下,应该可以使用同样的基本数据

服务来创建多个应用。数据层已经在第3章“管理数据”中有过详细讨论。



注意：有很多方式来表示软件设计，也有很多相关的书籍可供参考。现今，大多数行业使用名为统一建模语言(Unified Modeling Language, UML)的表示法。从本质上讲，它是类和结构以及对应的对象和它们相互作用的图形化表示。UML 是一个正式定义的设计语言。它最基本的功能是自动产生代码。它包括许多图和相关的图标。对于像本书中项目这样的小项目，UML 是不必要的开销，但是如果曾经在更大项目中工作并需要记录和分享程序的结构时，你应该研究 UML。

用户界面层、逻辑层和数据层之间的交互通常使用一个名为模型-视图-控制器(Model View Controller, MVC)的设计模式。大体上讲，模型代表核心逻辑层和数据层，而视图代表用户界面的显示元素，控制器代表这些显示元素之间的交互和依赖。在本章中，将使用一个简化版本的 MVC 模式来设计 GUI。



注意：MVC 模式本来是在 Xerox Parc 作为 Smalltalk 80 编程环境的一部分而开发的。多年以来，MVC 已经被许多不同语言和 UI 框架所采用。在这个过程中，它已经与 Smalltalk 的原始版本有了显著的区别。但核心思想仍然相同：将数据从展示(视图)和交互(控制器)中分离出来。

4.2 创建命令行界面

在本节中，将为著名游戏 tic-tac-toe 创建一个非常简单的命令行界面应用。本节会使用之前章节中讨论的原理，但形式非常简单，这样你就可以专注于程序结构而不是代码的功能等细节(代码在 Chapter4.zip 文件的 OXO 文件夹下)。

4.2.1 创建数据层

在创建这个游戏时，首先需要设计数据层。对于这个游戏，只需要一个简单的文本文件来保存游戏状态，这样它可以被保存和恢复使用。tic-tac-toe 游戏包含一个拥有 9 个正方形的面板，每个正方形中或者为空或者是 X 或 O 字符。可以使用简单的字符串列表来表示这三个选项。为了存储，将列表转换为一个简单的字符串。

需要的另一个数据是下一步是哪个玩家。但是在人机游戏中，可以假设人永远是下一步的玩家。所以你的数据层接口应提供或发布如下两个方法：


```
saveGame()
restoreGame()
```

既然你想要保持这些层独立，就应该把这些方法放入模块中。为了创建这个模块，在文件中输入下面的代码并保存为 `oxo_data.py`(或者从下载文件 `Chapter4.zip` 的 `OXO` 文件夹中载入它):

```
''' oxo_data is the data module for a tic-tac-toe (or OXO) game.
It saves and restores a game board. The functions are:
    saveGame(game) -> None
    restoreGame() -> game
Note that no limits are placed on the size of the data.
The game implementation is responsible for validating
all data in and out.'''
```

```
import os.path
game_file = ".oxogame.dat"
```

```
def _getPath():
    ''' getPath -> string
Returns a valid path for data file.
Tries to use the users home folder, defaults to cwd'''

    try:
        game_path = os.environ['HOMEPATH'] or os.environ['HOME']
        if not os.path.exists(game_path):
            game_path = os.getcwd()
    except (KeyError, TypeError):
        game_path = os.getcwd()
    return game_path
```

```
def saveGame(game):
    ''' saveGame(game) -> None
saves a game object in the data file in the users home folder.
No checking is done on the input, which is expected to
be a list of characters'''
```

```
path = os.path.join(_getPath(), game_file)
with open(path, 'w') as gf:
    gamestr = ''.join(game)
    gf.write(gamestr)
```

```
def restoreGame():
    ''' restoreGame() -> game

Restores a game from the data file.
The game object is a list of characters'''
```

```

path = os.path.join(_getPath(), game_file)
with open(path) as gf:
    gamestr = gf.read()
    return list(gamestr)

def test():
    print("Path = ", _getPath())
    saveGame(list("XO XO OX"))
    print(restoreGame())

if __name__ == "__main__": test()

```

第一个函数 `_getPath()` 是一个辅助函数。它使用 `os` 模块试图得到用户的主目录，如果失败就使用当前目录。按照惯例，不希望被模块使用者调用的函数会在名字前加一个下划线，例如 `_getPath()`。`saveGame()` 函数使用 `_getPath()` 创建了一个包含表示游戏的字符串的新文件。最后一个函数 `restoreGame()` 同样使用 `_getPath()` 定位到被保存的文件并打开它，读取保存的游戏数据。

理想情况下，你可能会包含一个更加复杂的测试函数(或一组单元测试，就像那些在第6章“Python 在更大项目中的应用”中描述的函数)。出于简洁性的考虑，在此不再赘述。



注意：在数据模块中，已经为模块和函数包含了文档字符串。应该永远在工作代码中这么做。但是由于代码的完整描述在文本中已提供并且为了节省本书的篇幅，文档字符串在其他模块中已被省略。

4.2.2 创建核心逻辑层

现在需要创建游戏的核心逻辑。为此，需要定义一些在游戏整个过程中用到的函数。然而，为了定义函数，首先需要考虑游戏是如何进行的。所以，在开始研究逻辑代码之前，需要先筹划游戏流程，为此，可以使用序列图来完成它。

游戏首先向用户展示一个选项菜单，它包含开始一个游戏或继续一个被保存游戏的选项。不论哪种情况，一旦游戏被设置，就会展示主面板并且提示用户选择一个格子。然后，计算机分析这个移动并以它的移动作为应答。之后，面板被再次展示出来直到一方获胜。序列图如图 4-1 所示。

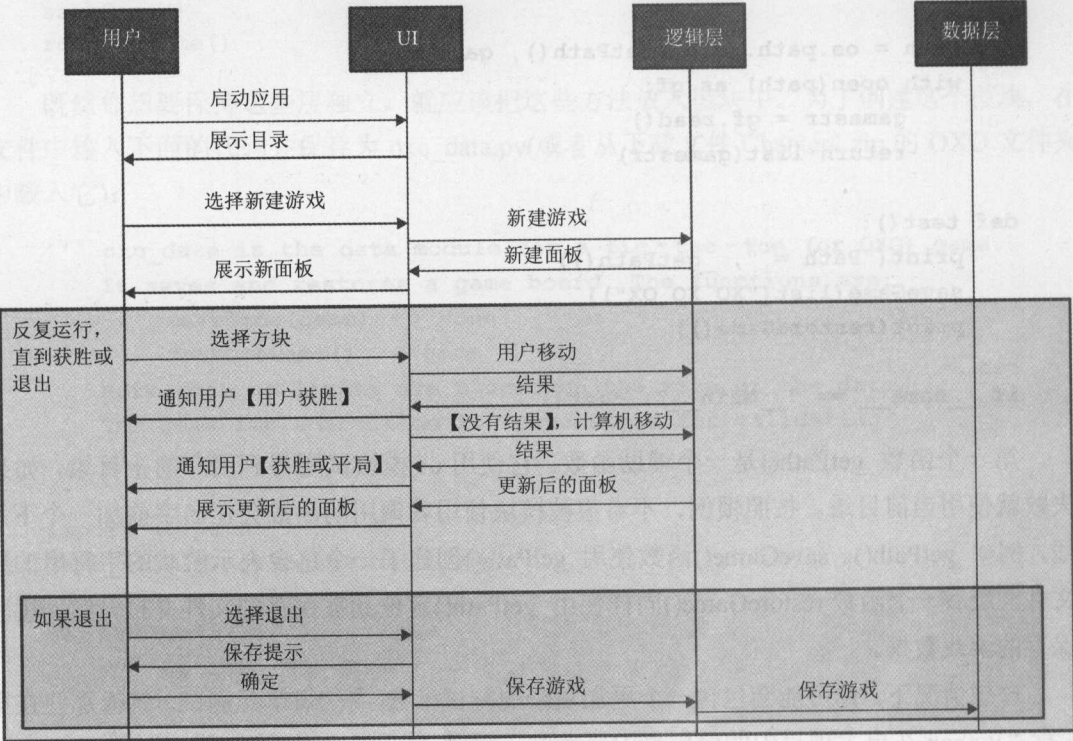


图 4-1 模块交互序列图

这个序列图是 UML 序列图的一个简化版本。每个对象(在这里是模块)和用户被表示为一条垂线。箭头表示两个模块(和用户)之间的消息流动。消息具有描述性的标题。一些消息是可选的，并且方括号中指明(被称为守卫)了产生这些消息的条件。一些箭头序列被包含在方框中。它们代表循环或条件选择块，它们的描述性信息在左上角。在本示例中，主游戏序列被一直重复直到一方获胜、平局或用户选择退出。如果用户选择退出，最底下的方框中的序列就会执行。序列图是一个非常强大的分析和设计工具。

从序列图中可以看到，需要提供函数来处理用户的菜单选择(图中只展示了新建菜单)以及一个玩游戏的函数，后者需要根据输出返回不同的结果。这通常是个糟糕的主意。然而，如果为用户和计算机的移动单独创建函数，之后就可以为用户和计算机使用同样的分析辅助函数。清楚了解流程之后，需要编写如下函数：

```
newGame ()
saveGame ()
restoreGame ()
userMove ()
computerMove ()
```

需要辅助函数来产生一个随机移动并分析某个给定的移动是否赢得了游戏。因此，辅助函数列表如下：

```
_generateMove ()
```



```
_isWinningMove()
```

同样，为了保持层之间的独立性，应该将逻辑代码放入一个单独的模块中，在此为 `oxo_logic.py`。代码如下：

```
''' This is the main logic for a tic-tac-toe game.
```

```
It is not optimised for a quality game it simply
generates random moves and checks the results of
a move for a winning line. Exposed functions are:
```

```
newGame()
```

```
saveGame()
```

```
restoreGame()
```

```
userMove()
```

```
computerMove()
```

```
'''
```

```
import os, random
```

```
import oxo_data
```

```
def newGame():
```

```
    return list(" " * 9)
```

```
def saveGame(game):
```

```
    oxo_data.saveGame(game)
```

```
def restoreGame():
```

```
    try:
```

```
        game = oxo_data.restoreGame()
```

```
        if len(game) == 9:
```

```
            return game
```

```
        else: return newGame()
```

```
    except IOError:
```

```
        return newGame()
```

```
def _generateMove(game):
```

```
    options = [i for i in range(len(game))
```

```
                if game[i] == " "]
```

```
    return random.choice(options)
```

```
def _isWinningMove(game):
```

```
    pass
```

`newGame()`函数只是返回一个拥有9个空格的新列表。

`saveGame()`函数调用同名的 `oxo_data` 函数。`restoreGame()`函数稍微有些复杂，因为它捕捉任何由于找不到数据文件所抛出的错误并验证保存游戏的长度。它可以在数据内容上做更多的数据有效性验证，但是如果愿意的话，可以在之后添加这些功能。

`_generateMove()`函数寻找当前游戏中没有被使用的格子，然后随机选择一个格子来防止计算机的移动。这不是最好做法，而更加智能的算法可以极大地提升游戏质量。

`_isWinningMove()`方法还没有完成，因为它是逻辑函数中最复杂的。这也是真正的处理开始的地方。你采用的方法非常简单并且依赖一个事实：只有 8 个可能的获胜路线。可以根据涉及的游戏格子的下标列出它们：

```
wins = ((0,1,2), (3,4,5), (6,7,8),
        (0,3,6), (1,4,7), (2,5,8),
        (0,4,8), (2,4,6))
```

为了评估一个移动是否赢得比赛，需要检查每一条获胜路线。可以提取候选路线中格子的字符并构建一个三字符的字符串。如果要赢得比赛，所有三个字符需要是 X 或者 O。函数如下：

```
def _isWinningMove(game):
    wins = ((0,1,2), (3,4,5), (6,7,8),
            (0,3,6), (1,4,7), (2,5,8),
            (0,4,8), (2,4,6))

    for a,b,c in wins:
        chars = game[a] + game[b] + game[c]
        if chars == 'XXX' or chars == 'OOO':
            return True
    return False
```

最后，需要一对可以分析用户移动和计算机移动的函数。前者接受用户输入的格子值，后者只需要游戏，因为它在内部使用 `_generateMove()`。它们把一个四字符的编码作为移动的结果返回。一个空字符串意味着游戏仍然继续，X 或 O 意味着获胜，而 D 意味着平局。这些函数如下：

```
def userMove(game,cell):
    if game[cell] != ' ':
        raise ValueError('Invalid cell')
    else:
        game[cell] = 'X'
    if _isWinningMove(game):
        return 'X'
    else:
        return ''
```

```
def computerMove(game):
    cell = _generateMove(game)
    if cell == -1:
        return 'D'
    game[cell] = 'O'
    if _isWinningMove(game):
        return 'O'
    else:
        return ''
```

应该已经实现了 `Game` 类中的所有这些方法。这可能已经消除了为每个函数传送游戏数据的需要。

最后，需要一个测试函数：

```
def test():
    result = ''
    game = newGame()
    while not result:
        print(game)
        try:
            result = userMove(game, _generateMove(game))
        except ValueError:
            print("Oops, that shouldn't happen")
        if not result:
            result = computerMove(game)

        if not result: continue
        elif result == 'D':
            print("Its a draw")
        else:
            print("Winner is:", result)
        print(game)

if __name__ == "__main__":
    test()
```

测试函数持续不断地生成移动，直到某一方赢得比赛或者面板被填满(此时，`_generateMove()`返回-1)。由于这些移动完全是随机生成的，因此在选择时没有任何智能。可以运行代码，并应该看到如下内容(实际得到的结果是随机的，因为你使用 `random` 模块来生成移动)：

```
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ]  
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', 'O', ' ', ' ', 'X', ' ', ' ' ]  
[ ' ', ' ', 'X', ' ', ' ', 'O', ' ', ' ', 'O', ' ', ' ', 'X', ' ', ' ' ]  
Winner is: X  
[ ' ', ' ', 'X', ' ', ' ', 'O', 'X', 'O', 'O', 'X', ' ', ' ' ]
```

它并不美观，也不应该美观。展示是在用户界面完成的，下面将创建用户界面。

4.2.3 创建用户界面

在上一节中，你开始思考用户会如何感知应用，也就是用户体验。用户体验是驱动用户界面设计的主要因素。需要考虑从进入应用、正常使用、到最后退出的控制流。在一个命令行应用中，最常见的方法是提供菜单。菜单通常会嵌套几层，或者接受在提示符输入的命令。在本节中，将创建一组非常简单的菜单并且创建一个简单的基于提示的解决方案。这个过程在阐述这两种技术的同时，还保持了代码的简洁性。

如果熟悉 tic-tac-toe 游戏, 就知道它首先提供了一个菜单。这个菜单允许用户启动一

个新游戏、继续一个保存的游戏、请求帮助或退出(能够轻松退出是优秀用户界面设计中非常重要但经常被忽略的一个特性)。

退出选项只是退出程序，帮助选项会展示一页解释的文字，而另外两个选项将用户带入 tic-tac-toe 游戏。在该游戏中，用户可以选择一个格子进行移动、保存游戏或退出游戏。如果用户在游戏进行中选择退出游戏，那么应该首先要询问他们是否想要保存游戏。如果他们选择一个格子进行移动，那么计算机会分析移动，做出它自己的移动，分析这个移动并且展示结果。如果任何一个移动导致获胜或所有格子被使用，就为用户显示一个消息和一个用于退出或返回主目录的选项。

用户界面用于显示信息并管理屏幕之间的切换。它不执行任何计算逻辑，这是由核心逻辑层提供的。用户界面只会调用在 `oxo_logic` 模块中定义的函数。

首先会定义一个函数来展示菜单并返回一个有效的用户选择。函数可以特定于菜单并在它内部包含菜单定义，编写一个接受菜单作为输入的函数比较简单，而且还可以复用于系统中的所有菜单。甚至可以将它提取出来形成一个模块，这样可以在项目间复用。

`oxo_ui.py`(或从 `zip` 文件中加载它)中的菜单代码如下：

```
''' CLI User Interface for Tic-Tac-Toie game.
    Use as the main program, no reusable functions'''

import oxo_logic

menu = ["Start new game",
        "Resume saved game",
        "Display help",
        "Quit"]

def getMenuChoice(aMenu):
    ''' getMenuChoice(aMenu) -> int

        takes a list of strings as input,
        displays as a numbered menu and
        loops until user selects a valid number'''

    if not aMenu: raise ValueError('No menu content')
    while True:
        print("\n\n")
        for index, item in enumerate(aMenu, start=1):
            print(index, "\t", item)
        try:
            choice = int(input("\nChoose a menu option: "))
            if 1 <= choice <= len(aMenu):
                return choice
            else: print("Choose a number between 1 and", len(aMenu))
        except ValueError:
            print("Choose the number of a menu option")
```

```
def main():
    print(getMenuChoice(menu))
    getMenuChoice([]) # raise error

if __name__ == "__main__": main()
```

注意，enumerate 函数被用于生成菜单项数字，而它们开始于 1，这是因为大多数用户认为 0 是一个奇怪的选项。函数会一直重复执行，直到用户做出一个有效的选择，同时提示符号纠正无效选择。

在展示了一些选项后，现在需要编写代码完成所有事情。方法名是以菜单选项命名的，如下所示：

```
def startGame():
    return oxo_logic.newGame()

def resumeGame():
    return oxo_logic.restoreGame()

def displayHelp():
    print('''
    Start new game: starts a new game of tic-tac-toe
    Resume saved game: restores the last saved game and commences play
    Display help: shows this page
    Quit: quits the application
    ''')

def quit():
    print("Goodbye...")
    raise SystemExit
```

下一步是编写一个函数来处理用户的选择。这可以使用一个 if/elif 结构来实现，但是如果有很多选项，这会变得非常难以维护。尽管这个项目可能的选项数量很小，但还是使用了一个分派表，因为这是一种强大、高效且灵活的技术。也需要修改 main() 函数来遍历菜单和游戏代码直到用户退出。对程序做如下修改：

```
def executeChoice(choice):
    ''' executeChoice(int) -> None

    Execute whichever option the user selected.
    If the choice produces a valid game then
    play the game until it completes.'''
    dispatch = [startGame, resumeGame, displayHelp, quit]
    game = dispatch[choice - 1]()
    if game:
        # play game here
        pass

def main():
```

```

while True:
    choice = getMenuChoice(menu)
    executeChoice(choice)

```

现在只剩下实际玩游戏的任务。需要编写一个函数，它接受一个开始的游戏位置作为参数，并与用户交互直到游戏结束。游戏结束可能是因为没有更多可移动的格子，也可能是因为找到了获胜者。此外，如果有一个函数以普通的网格布局而不是内部正在使用的平面文件格式展示游戏，那么它会很有用。如下所示：

```

def printGame(game):
    display = '''
    1 | 2 | 3      {} | {} | {}
    -----
    4 | 5 | 6      {} | {} | {}
    -----
    7 | 8 | 9      {} | {} | {}'''
    print(display.format(*game))

def playGame(game):
    result = ""
    while not result:
        printGame(game)
        choice = input("Cell[1-9 or q to quit]: ")
        if choice.lower()[0] == 'q':
            save = input("Save game before quitting?[y/n] ")
            if save.lower()[0] == 'y':
                oxo_logic.saveGame(game)
            quit()
        else:
            try:
                cell = int(choice) - 1
                if not (0 <= cell <= 8): # check range
                    raise ValueError
            except ValueError:
                print("Choose a number between 1 and 9 or 'q' to quit ")
                continue

            try:
                result = oxo_logic.userMove(game, cell)
            except ValueError:
                print("Choose an empty cell ")
                continue

            if not result:
                result = oxo_logic.computerMove(game)
            if not result:
                continue
            elif result == 'D':
                printGame(game)
                print("Its a draw")

```



```

else:
    printGame(game)
    print("Winner is", result, "\n")

```

`printGame()`函数将游戏数据格式化之后展示出来。注意它使用了星号(`*game`)来扩展到列表的每个单独的元素。

`playGame()`函数在游戏屏幕上展示用户界面提示。要求一个代表用户希望放入 X 的格子的数字。也提供了退出选项。当用户选择退出时，程序会提示他们保存游戏。如果用户的选择有效，就使用 `oxo_logic` 函数来得到选择的结果。但是如果无效，就结束移动而轮到计算机。注意所有游戏规则和数据管理是如何放在下一层处理的。用户界面层只处理展示和流控制。一个设计的怪事是面板被传入 `userMove()`和 `computerMove()`函数中，但是函数没有返回更新后的面板。这是因为面板对象是一个可变列表，它可以被函数更改。函数对初始变量的改变将会在面板上反映出来。

在本章后面部分，会再次访问这个游戏并且在现有的 `oxo_logic` 和 `oxo_data` 模块上轻松创建一个新的基于 GUI 的用户界面层。在此之前，还有几个可用于增强命令行应用的有意思的选项。接下来将介绍它们。

4.3 使用 cmd 模块创建命令行界面

Python 标准库中有一个 `cmd` 模块，专门用来创建命令行界面。事实上，它创建了 Python 的帮助和调试的系统界面，展示了一个命令提示符。可以输入一个命令，并请求帮助。此时会展示一个帮助屏幕。如果输入 `help<command>`，会得到关于此命令的帮助。

在本节中，会创建一个 tic-tac-toe 游戏。它是一个基于 `cmd` 的版本。在开始菜单中，它同样有 4 个选项。玩游戏的部分和之前完全一样(完整代码在 Chapter4.zip 文件的 OXO 文件夹的 `oxo-cmd.py` 文件中)。

`cmd` 基于一个面向对象的框架。在此基础上，定义了一个 `cmd.Cmd` 的子类。它重写了一些关键方法以支持应用的行为。然后，定义一组以 `do_` 开头的方法。这个类将下划线后面的部分解释为用户可以输入的命令。

可以通过定义一些打印消息的方法来构建 tic-tac-toe 游戏的骨架。通过它，可以看到它的工作原理。代码如下：

```

import cmd

class Oxo_cmd(cmd.Cmd):
    intro = "Enter a command: new, resume, quit. Type 'help' or '?' for help"
    prompt = "(oxo) "
    def do_new(self, arg):
        print("Starting new game")

    def do_restore(self, arg):
        print("Restoring previous game")

```

```

def do_quit(self, arg):
    print("Goodbye...")
    raise SystemExit

def main():
    game = Oxo_cmd().cmdloop()

if __name__ == "__main__":
    main()

```

通过上述代码，创建了一个继承 `cmd.Cmd` 的新类。为它添加了一个介绍信息和一个提示字符串。然后，定义了命令-响应方法。

注意，不需要一个帮助命令，因为它已经被自动创建在类机制中了。注意还需要在方法定义中提供第二个虚拟参数，即使在方法中没有使用它。

最后，在 `main` 函数中，实例化 `game` 对象并执行 `cmdloop()` 方法。

如果运行它，就会看到它产生了一个全功能的命令行解释器应用。

为了把它转变为可以工作的 tic-tac-toe 游戏，只需要修改几个小地方。导入 `oxo_logic` 和 `oxo_ui` 模块。创建 `game` 类变量来保存游戏数据。然后，从命令方法中调用 `ui_logic` 模块函数。最后，调用 `oxo_ui.playGame()` 方法初始化游戏。

最后的游戏代码如下：

```

import cmd, oxo_ui, oxo_logic

class Oxo_cmd(cmd.Cmd):
    intro = "Enter a command: new, resume, quit. Type 'help' or '?' for help"
    prompt = "(oxo) "
    game = ""

    def do_new(self, arg):
        self.game = oxo_logic.newGame()
        oxo_ui.playGame(self.game)

    def do_resume(self, arg):
        self.game = oxo_logic.restoreGame()
        oxo_ui.playGame(self.game)

    def do_quit(self, arg):
        print("Goodbye...")
        raise SystemExit

def main():
    game = Oxo_cmd().cmdloop()

if __name__ == "__main__":
    main()

```

可以探索 `cmd.Cmd` 类的许多其他选项和特性，但希望这个示例已经能够向你演示如何

轻松使用 `cmd` 创建一个命令行解释器风格的应用。它应该也证明了展示和逻辑分开是如何的强大。仅仅使用了少量的代码，你就创建了一个完全不同版本的 `tic-tac-toe` 游戏，但是逻辑层和数据层是完全相同的。

在下一节中，你会了解命令行以及如何读取命令行输入参数。

4.4 读取命令行参数

在启动命令程序时，命令行本身作为字符串列表被存储在 `sys.argv` 中。第一个元素是脚本名称，而接下来的元素是命令的参数。因此，如果有一个文件拷贝脚本，你可能想按如下方式调用它：

```
$ python mycopy.py originalfile copyfile
```

而 `sys.argv` 值应该是：

```
["mycopy.py", "originalfile", "copyfile"]
```

然而，通常的情况是命令行脚本接受可选参数来控制显示或功能。例如，许多程序提供一个 `-h` 或 `--help` 选项来显示命令的帮助信息。可以通过检查 `sys.argv` 的内容来处理这些选项，但是这并不轻松。所以 Python 提供了 `argparse` 模块来辅助处理这些命令选项。

现在要修改初始的 `tic-tac-toe` 代码，让它在命令行出现 `-h` 或 `--help` 时显示帮助信息。如果 `-n` 或 `--new` 被指定，它会直接转向一个新游戏，而如果给定的是 `-r` 或 `--res` 或 `--restore`，它会直接继续游戏。如果 `tic-tac-toe` 玩家有经验并且愿意的话，这可以帮助他们绕过初始菜单。



注意：Python 的参数处理会假定参数是 UNIX 风格的——以一个或两个连字符开始。DOS 或 Windows 上的传统选项风格是斜杠。 `argparse` 默认不处理那些参数，但是如果需要支持其他风格，可以在创建解析器时指定前缀字符为可选参数。

首先需要导入模块：

```
import argparse as ap
```

接下来修改主函数：

```
def main():
    p = ap.ArgumentParser(description="Play a game of Tic - Tac - Toe")
    grp = p.add_mutually_exclusive_group()
    grp.add_argument("-n", "--new", action='store_true',
```



```

        help="start new game")
    grp.add_argument("-r", "--res", "--restore", action='store_true',
                    help="restore old game")
    args = p.parse_args()

    if args.new:
        executeChoice(1)
    elif args.res:
        executeChoice(2)
    else:
        while True:
            choice = getMenuChoice(menu)
            executeChoice(choice)

```

现在，试一试-h(或--help)选项，它会生成一个由 argparse 生成的标准格式的帮助用户。

```
usage: oxo_args_ui.py [-h] [-n | -r]
```

```
Play a game of Tic-Tac-Toe
```

```
optional arguments:
```

```

-h, --help            show this help message and exit
-n, --new             start new game
-r, --res, --restore  restore old game

```

如果指定-n(或--new)，它直接创建一个新游戏；如果指定-r、--res 或--restore，则回到最后保存的游戏。

注意 argparse 代码中的一些特性。首先，创建了一些作为互相独立的组的选项。那是因为在同一个命令中同时指定新建和继续选项是毫无意义的。同样，在 ArgumentParser 的构造函数中提供了程序的描述，这些信息之后会出现在帮助屏幕中。最后，通过指定 action 的值为 store_true，将选项变成了布尔值，这样它们可以被当作真值用在 if/elif 测试中。

argparse 模块有很多其他功能。它可以把参数解释为不同类型，可以为选项计数，还可以关联多个选项等。有一个指南介绍了有关该模块的用法，并且文档中也给出了几个相关示例。

最后需要指出的是：在没有对游戏逻辑层或数据层做任何修改的情况下，已经高效地创建了另一个用户界面。将展示与逻辑和数据分开是一种非常强大的设计技术。

在下一节中，将介绍另一种创建命令行应用的方式。新应用添加了一些 GUI 特性但并没有实现一个完整的 GUI 应用，新的外观会让用户感到更加专业。

4.5 用一些对话框让命令行界面变得生动

不需要构建完整的图形用户界面，也可以为命令行界面应用添加一些 GUI 元素。例如，

可以弹出信息或警告框而不是简单地在终端上打印信息。这通常让消息在用户面前更加突出，它们不会被淹没在屏幕的茫茫文本中。也可以在选择文件名时使用标准文件选择对话框。在本节中将为 tic-tac-toe 用户界面添加一些 GUI 消息框来突显错误消息并通知用户游戏的最终结果。



提示：让代码也输出到终端是个很好的主意。有可能它们并没有运行 GUI 环境，比如用户使用 ssh 或类似的方式远程访问。你当然可以这样做，也可以提供一个命令行选项，例如 `--nogui`，来控制提示信息的显示。

在开始修改游戏本身之前，可以研究一下这些消息框在命令行提示符上的工作方式，它们被定义在 `tkinter` 包的子模块中。你将在本章后面更全面地研究它。这些子模块如下：

- `tkinter.messagebox`
- `tkinter.filedialog`
- `tkinter.simpledialog`
- `tkinter.colorchooser`
- `tkinter.font`

在本节中，你只使用了第一个模块，但是对于所有模块来说，原理是相同的。

遗憾的是，官方文档极少，所以在 Python 提示符上做一些实验会大有裨益。现在就可以使用 `tkinter.messageBox` 模块试一试。

试一试：探索 Tkinter 消息框

在这个“试一试”中，将使用 Python 的 `tkinter.messagebox` 模块中的各种可用的消息框。将学习如何把这些集成到一个非 GUI 程序中，以及如何处理一些琐事。完成下面的步骤：

(1) 启动 Python 解释器并输入下面的代码：

```
>>> import tkinter.messagebox as mb
>>> mb.showinfo("Title", "Your message here")
'ok'
>>>
```

结果如图 4-2 所示。注意，消息框中包含标题和消息，以及表明它是一个信息消息的图标。同时，出现了奇怪的空窗口，你想要隐藏它。

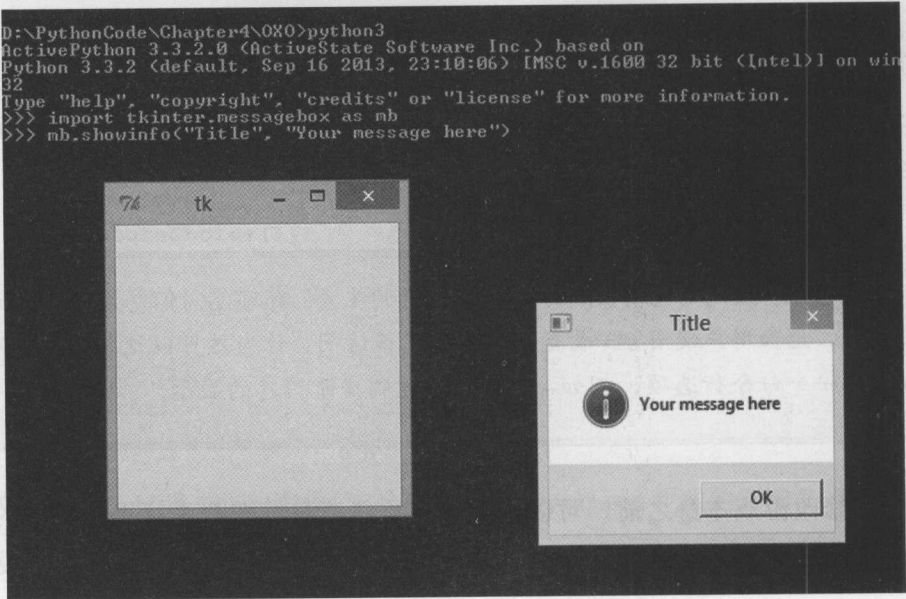


图 4-2 初始消息框的屏幕截图

(2) 为了去掉空窗口，关闭两个已显示出来的 Tkinter 窗口并输入下面的代码：

```
>>> import tkinter
>>> tk = tkinter.Tk()
>>> tk.withdraw()
''
```

(3) 注意空窗口又出现了。一旦输入命令 `withdarw()`，它就会消失。

(4) 现在，可以通过下面的代码弹出消息框：

```
>>> mb.showinfo("Title", "That's better!")
'ok'
>>>
```

(5) 关闭对话框并试一些其他的消息框，如下所示：

```
>>> dir(mb)
['ABORT', 'ABORTRETRYIGNORE', 'CANCEL', 'Dialog', 'ERROR', 'IGNORE',
 'INFO', 'Message', 'NO', 'OK', 'OKCANCEL', 'QUESTION', 'RETRY',
 'RETRYCANCEL', 'WARNING', 'YES', 'YESNO', 'YESNOCANCEL',
 '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_show', 'askokcancel', 'askquestion', 'askretrycancel',
 'askyesno', 'askyesnocancel', 'showerror', 'showinfo',
 'showwarning']
>>> mb.showerror("An Error", "Oops!")
'ok'
>>> mb.showwarning("Title", "This may not work...")
'ok'
>>> mb.askyesno("Title", "Do you love me?")
```



```

True
>>> mb.askokcancel("Title", "Are you well?")
True
>>> mb.askquestion("Title", "How are you?")
'yes'
>>> mb.askretrycancel("Title", "Go again?")
True
>>> mb.askyesnocancel("Title", "Are you well?")
>>>

```

示例说明

首先，导入了 `messagebox` 子模块并为其起了一个别名，以减少输入。然后试图显示一个错误，但是发现除了期待的信息消息又出现了第二个空窗口。实际上，第二个窗口是主 Tkinter 应用窗口。通常，你会将控件、菜单等放在这个窗口。为了不让它出现，必须导入主 Tkinter 模块并实例化顶层 Tk 对象。然后通过调用 `withdraw()` 让对象不可见。完成这些之后，可以随时调用 `messagebox` 对象，而顶层窗口会一直不可见。

然后，更深入地探索了模块。通过 `dir()` 列表，可以看到其他可用的函数并轮流试试它们。注意，一些函数返回字符串，比如 `ok`，而其他函数则返回布尔结果。最好在交互式提示符上实验它们，以便知道返回值的类型。注意，当出现 `Cancel` 时，单击它会返回 `None`。

可以用同样的方式实验一些其他标准对话框，比如文件对话框或字体和颜色选择器。

在了解标准对话框的工作原理后，可以将它们用在 `tic-tac-toe` 游戏中，使用 `showinfo` 对话框显示结果。你也使用了一个 `askyesno` 对话框来询问用户在退出前是否想要保存当前游戏。尽管可以使用对话框生成所有的提示，但这实际上会导致用户界面非常累赘，界面上的对话框会弹出并立刻消失。

需要在 `playGame()` 函数中做一些修改。可以修改之前任意版本的游戏，但是在这个示例中，将以原始的 `oxo_ui.py` 文件作为基础并把它保存为 `oxo_dialog_ui.py` (或只是从 `Chapter4.zip` 文件加载它)。

首先，添加 `import` 语句：

```

import tkinter
import tkinter.messagebox as mb

```

然后，修改 `main`，消除顶层窗口：

```

def main():
    top = tkinter.Tk()
    top.withdraw()
    while True:
        choice = getMenuChoice(menu)
        executeChoice(choice)

```

最后，修改 `playGame()`，如下所示：

```

def playGame(game):
    result = ""
    while not result:
        printGame(game)
        choice = input("Cell[1-9 or q to quit]: ")
        if choice.lower()[0] == 'q':
            save = mb.askyesno("Save game", "Save game before quitting?")
            if save:
                oxo_logic.saveGame(game)
                quit()
        else:
            try:
                cell = int(choice)-1
                if not (0 <= cell <= 8):
                    raise ValueError
            except ValueError:
                print("Choose a number or 'q' to quit")
                continue

            try:
                result = oxo_logic.userMove(game, cell)
            except ValueError:
                mb.showerror("Invalid cell", "Choose an empty cell")
                continue

            if not result:
                result = oxo_logic.computerMove(game)
            if not result:
                continue
            elif result == 'D':
                printGame(game)
                mb.showinfo("Result", "It's a draw")
            else:
                printGame(game)
                mb.showinfo("Result", "Winner is {}".format(result))

```

尽管这里展示了很多代码，但只修改了少数几行。同样，逻辑层或数据层不需要修改。下一节将带你进入 GUI 的世界。

4.6 使用 Tkinter 编程 GUI

本节将介绍如何使用 Python 的标准 GUI 工具包 Tkinter 来创建 GUI。所有 GUI 都构建在函数工具包或更常见的类库上。稍后将介绍 Python 可用的其他工具包。现在，Tkinter 工具包为这些基本原理提供了坚实的基础。

首先，将研究 GUI 设计的一些基本概念，包括如何组织和使用 GUI 工具包。

主要 GUI 原理简介

实际上，所有 GUI 都是事件驱动的。这意味着需要编写代码来响应 GUI 工具包产生的特定事件。GUI 首先拥有一套它们自己的以对象为表现形式的语言。GUI 就是由这些对象创建的。对象包括窗口、框架、控件等。这些对象都是由名为包含树(containment tree)的东西连接的。在接下来的小节中，你将了解每个概念以及它们如何组装在一起来形成 GUI。

1. 基于事件的编程

在第 2 章“Python 脚本”中，介绍了如何用事件驱动程序。其中探索了 XML 和 HTML 文件的解析。在本质上，解析器使用了一个内部循环，无论何时它们碰到一个感兴趣的项，都会给代码发送一个信息。实际上，它们调用了提供的函数。



注意：这种函数类型有时被称为回调，因为在框架中注册了它，然后框架又调用了它。GUI 编程广泛地使用回调函数。

GUI 程序的作用方式与之类似。工具包内部有一个无限循环，而当用户单击按钮、移动鼠标或按键时，工具包会产生事件从而调用函数。编写函数并把它们注册到特定的事件上，这样当用户选择类似 File | Save 的菜单项时，就会调用函数 doFileSave()。

这意味着程序代码的外观发生了改变。不同于自始至终的控制程序运行，初始化你的数据，之后将控制移交给工具包。这对于一些程序员来说是一个令人不安的经历。但是一旦适应了它，就会发现它实际上将你从大量单调的控制流编程中解放出来并让你专注于程序需要做的事情。

2. GUI 术语

在处理 GUI 时，你会首先注意到需要学习的新术语的数目。你已经接触到了事件，并且毫无疑问地会接触到更多，比如菜单、按钮、滚动条等。作为程序员，你会发现有时这些术语的常识性理解并不是它在编程中的意思。此外，还有一堆用户通常见不到的术语。表 4-1 列出了一些最重要的 GUI 术语和它们对程序员的意义。

表 4-1 GUI 术语

术 语	描 述
窗口	应用控制的一个屏幕区域。窗口通常是矩形的，但是一些 GUI 环境允许其他形状。窗口可以包含其他窗口，通常，每个单独的 GUI 控件都被当作它本身的一个窗口对待
控件	控件是一个 GUI 对象，用来控制应用。控件有属性并且通常产生事件。通常，控件对应着应用层对象，而事件可以与对应对象的方法耦合在一起，这样当某个事件发生时，对象会执行它的一个方法

(续表)

术 语	描 述
小部件	一个可见的控件。一些控件(比如计时器)可以与一个给定窗口关联在一起，但仍然不可见。小部件是那些可见并且可以被用户或程序员操纵的控件子集
框架	一种用来将其他小部件集合在一起的小部件类型。经常用一个框架表示完整的窗口，其中还可以嵌入更多的框架。框架有时有可见的边缘和背景颜色，但有时只是作为一个容器对象而不可见
标签	一个包含文本或简单图片的小部件。它不产生任何事件，但是可以响应其他事件而改变
按钮	一个带有文字和/或图片的小部件。用户可以单击它，产生一个事件作为响应
文本输入框	可以显示和/或接收文本的小部件。它可以是窗体上的单行输入，或者多行输入，比如一个文本编辑器窗格。文本小部件通常可以包含其他小部件，比如图片
菜单	显示菜单控件的小部件。菜单包含菜单项和/或子菜单。菜单提供所有用于遍历菜单小部件层级的机制。当选择菜单项时，会引发一个事件
画布	包含图形形状和图片的小部件。画布对象通常包含绘制几何图形、表等的方法
几何结构	每个窗口和小部件都有一个几何结构或用来表示它位置和大小坐标组。不同的工具包用不同的方式显示这个信息。Tkinter 使用格式(宽, 高)。如果需要的话，位置信息被显示为: (x 坐标, y 坐标)并且是相对于包含它的小部件的
对话框	父应用拥有的一种特殊类型的窗口。但是它可以独立地在屏幕四周移动。对话框可以是模态的，这意味着在应用响应任何其他动作前必须关闭对话框。也可以是非模态的。此时，对话框与主应用窗口是并行的
消息框	一个小对话框。通常用来表示非常简单的提示或请求简单类型的用户输入。已经在前面使用过 Tkinter 的消息框
布局	框架中的控件是根据一组特殊的规则或指导方案显示的。这些规则就是布局。可以通过多种方式指定布局，或者使用屏幕上以像素为单位的坐标，或者使用相对于其他组件的相对位置(左, 上等)，或者使用一个网格或表格布置。坐标系统很好理解，但是在比如窗口大小被调整时很难管理。在使用基于坐标的布局时，建议使用不可调整大小的窗口。最好使用非坐标布局，而让工具包为你管理这些事情
父-子	GUI 应用倾向于包含一个小部件/控件层级。容纳应用窗口的顶层框架包含子框架。这些子框架同样也包含更多的框架或控件。这些控件可以被看作树型结构。每个控件有唯一的父控件和多个子控件。实际上，这个结构通常被小部件显式地存储起来，这样程序员或更常见的 GUI 环境本身可以经常对一个控件及其所有子控件执行相同的行为。例如，关闭最顶层小部件也会关闭它所有的子小部件。包含其他小部件的子控件被称为父控件
焦点	当一个窗口获得焦点时，它就变成了活动窗口，因为所有按键和鼠标单击将会作用在那个窗口和它所有的子控件上。例如，一个文字处理器可能有一个用于搜索的非模态的对话框。用户可以在主窗口和对话框之间通过鼠标单击任何接受输入的窗口来切换焦点

3. 包含树

每个 GUI 应用都是用类树的方式构建的，有一个包含其他窗口的顶层窗口，其他窗口可以包含更多窗口，直到你到达最底层的控件和小部件。这个层级可以被表示为一个树型结构，并且也被称为包含树。

事件会到达一个子小部件，如果子小部件不能处理它，就会把事件向上传给它的父控件直到顶层。同样，如果有一个命令是绘制一个子小部件，它会向下把命令发送到它的子控件。因此，一个顶层小部件的绘制命令重新绘制了整个应用。不过，发送到按钮的绘制命令只会重新绘制按钮。

所以，事件是自下而上地渗透整个树，而命令是自上而下的。这个概念对于理解 GUI 在程序员层如何操作是最基本的。这也是为什么你经常需要在创建小部件时指定它的父控件的原因所在。这样就知道它处于包含树的什么位置了。

一个包含树示例如图 4-3 所示。

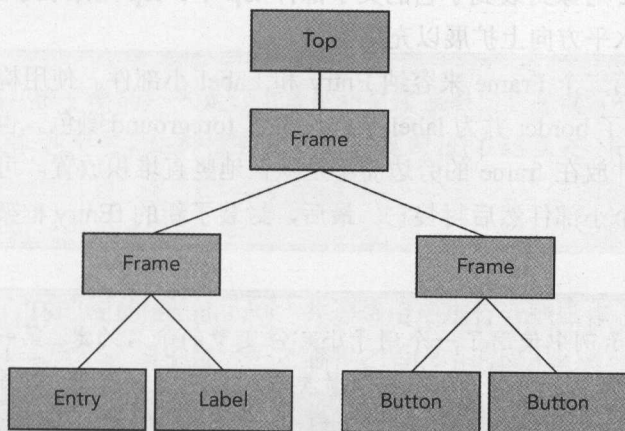


图 4-3 一个 GUI 包含树的示例

该图演示了顶层小部件包含了一个单独的 Frame，这个 Frame 代表最外面的窗口边缘，它包含了两个额外的 Frame。第一个 Frame 含有一个文本 Entry 小部件和一个 Label。而第二个 Frame 含有两个用于控制应用的 Button。在下一节中，当准备好创建一个简单的 GUI 时，你应该回顾一下这个图。

4.7 创建简单的 GUI

现在我们将上述讨论变成真正的代码。首先，创建如图 4-3 所示的应用。代码如下(可以从 Chapter4.zip 的 Tkinter 文件夹中的 demo1.py 文件加载)：

```
import tkinter as tk

# create the top level window/frame
top = tk.Tk()
```

```

F = tk.Frame(top)
F.pack(fill="both")

# Now the frame with text entry
fEntry = tk.Frame(F, border=1)
eHello = tk.Entry(fEntry)
eHello.pack(side="left")
lHistory = tk.Label(fEntry, text=" ", foreground="steelblue")
lHistory.pack(side="bottom", fill="x")
fEntry.pack(side="top")

```

首先，导入了 `tkinter` 模块并创建了一个顶层小部件(在之前使用消息框的部分，已经见过它)。接下来，创建了一个 `Frame` 来容纳所有其他小部件。所有小部件创建方法的第一个参数都是父小部件。所以在示例中，指定父小部件为 `top`。下一步是调用 `F.pack()`。调用 `pack()` 触发了一个简单的布局管理器。在默认情况下，它只是自上向下地将组件封装到包含对象中。此时，把 `Frame` 对象封装到了它的父小部件 `top` 中。`top` 用来表示主窗口。`fill` 选项告诉小部件在垂直和水平方向上扩展以充满窗口。

然后，创建了第二个 `Frame` 来容纳 `Entry` 和 `Label` 小部件。使用构造函数的命名参数为 `entry` 小部件设置了 `border` 并为 `label` 字体设置了 `foreground` 颜色。也要注意，使用参数告诉包装器将小部件放在 `frame` 的旁边而不是默认地竖直堆积放置。可以看到模块化开发的用法——创建一个小部件然后封装它。最后，封装了新的 `fEntry` 框架。



注意：示例中使用了一个用于小部件变量的命名约定。第一个字符表明了小部件的类型。这有时作为每个变量类型的提醒器很有用，但是如果之后改变小部件的类型，就会产生维护的问题。使用这个命名约定是完全可选的，对于 `Python` 或 `Tkinter` 来说没有任何区别。

下一步是创建按钮并将它们与一些行为关联起来。需要创建一个事件处理程序。当用户按下按钮时，它就会被激活：

```

# create the event handler to clear the text
def evClear():
    lHistory['text'] = eHello.get()
    eHello.delete(0, tk.END)

```

事件处理程序将 `lHistory` 标签的文本设置为 `eHello` 输入区域的内容，然后将 `eHello` 区域的文本删除。你使用了字典风格的访问方式设置了标签的文本。这种技术也同样适用于小部件的任意属性。`delete()` 方法将 `0` 作为第一个参数，这表明文本的开始，而第二个参数是 `tk.END`，它是一个特殊值，意味着文本的末端。

使用下面代码创建按钮并连接到事件处理程序：


```

# Finally the frame with the buttons.
# sink this one for emphasis
fButtons = tk.Frame(F, relief="sunken", border=1)
bClear = tk.Button(fButtons, text="Clear Text", command=evClear)
bClear.pack(side="left", padx=5, pady=2)
bQuit = tk.Button(fButtons, text="Quit", command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)
fButtons.pack(side="top", fill="x")

# Now run the eventloop
F.mainloop()

```

再次创建了一个框架来容纳按钮，然后使用 `side` 参数封装了按钮。也为按钮添加了一些补白距以便在它们周围创建一些空白。通过指定函数名 `evClear` 作为命令参数，将 `bClear` 按钮连接到事件处理程序上。`bQuit` 按钮使用了在顶层框架 `F` 中预先定义的 `quit` 方法。



警告：很重要的一点是，你只是指定了函数名，并没有使用名称后加圆括号的方式来调用它。这样做会将函数的返回值(在这里是 `None`)赋值为事件处理程序。

最后，运行了 Tkinter 的 `mainloop()`，并等待用户进行一些操作。如果运行代码，产生的窗口应该如图 4-4 所示。图中显示了之前和之后版本的应用。

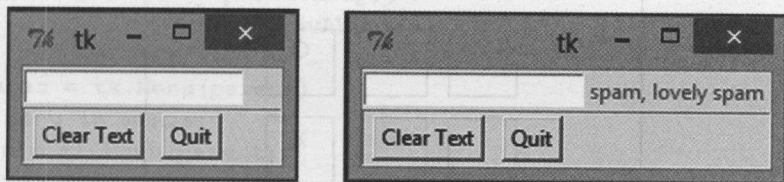


图 4-4 运行中的 `demo1.py`

你会发现可以像处理桌面上的其他窗口那样来调整大小、图标化并移动窗口。单击 `Quit` 按钮会关闭应用(就像在标题栏单击常用的关闭图标)，而单击 `Clear` 按钮会调用 `evClear` 函数。

再回到前面一节介绍的包含树图，可以看到它描述了这个应用的布局，它拥有三个框架、一个标签、一个输入框和两个按钮。

前面介绍了一些基础知识，接下来介绍一个更加实际的示例。它将使用更多的小部件并将控件链接到更有实质意义的事件处理函数上。是时候再次讨论 `tic-tac-toe` 游戏了。

4.8 创建 Tic-Tac-Toe GUI

在本节中，会使用与之前完全一样的逻辑层和数据层为 tic-tac-toe 游戏创建 GUI。这个 GUI 将会非常接近你期待在现代桌面应用上看到的 GUI 的样子，包括菜单、按钮和鼠标交互而不是依赖于键盘输入。通过使用与之前一样的 tic-tac-toe 逻辑，可以聚焦在 GUI 的结构上，而不必太多思考应用本身的工作方式。

4.8.1 勾勒一个 UI 设计

在创建一个有意义的 GUI 应用时，在编写代码之前粗略地勾勒出你想要的应用的外观通常很有帮助。对于 tic-tac-toe 游戏，你想要包含 File 和 Help 的菜单栏。File 菜单包含 New、Resume、Save 和 Exit 菜单项(你可能已经将菜单称为 Game 菜单，但是 File 是 GUI 应用的一个常用选项，而风格的一致性是使用 GUI 的一个优点)。Help 菜单包含 Help 和 About 选项。

面板本身由 9 个按钮以普通网格的风格布局。当一个按钮被单击时，它的标签会显示玩家的标记。

底端的一个状态栏会为用户显示消息，而最终结果会使用消息框显示出来，如图 4-5 所示。

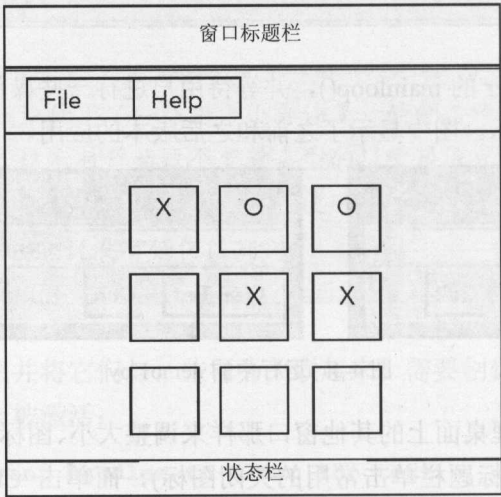


图 4-5 Tic-tac-toe GUI 设计

在对布局有了清晰认识之后，你现在想要开始思考代码的结构。GUI 本质上包含三个区域：菜单栏、面板和状态栏。面板需要被当作一个组来对待并且要放在框架的中心，这样可以将面板本身作为位于中心框架中的另一个框架。状态栏只显示文本，所以它仅会包含一个标签小部件。现在就剩下菜单栏了。

4.8.2 创建菜单

相比于你之前已经用过的子控件，Tkinter 菜单要稍微复杂些。创建初始的菜单栏实际上很简单，因为它是新菜单的默认样式。但是，如何创建下拉菜单呢？答案是使用菜单类中的一个名为 `add_cascade()` 的方法。它将一个子菜单应用到上层菜单中。Tkinter 了解菜单，所以它会自动创建必要的下拉和弹出菜单而不需要你做任何事情。最后一个菜单的差异之处是它们没有被普通的布局管理器方法(比如 `pack()`)添加到窗口中。相反，顶层小部件的菜单属性被设置为顶层菜单对象。

创建菜单会因为大量重复输入而变得非常无聊，所以更简单的方法是将菜单模型化为数据，然后使用循环将数据结构处理为菜单层级。尽管你的菜单实际上很短，但是也可以使用这个方法。

创建菜单的代码如下(可以从 zip 文件 OXO 文件夹加载 `oxo_menu.py` 文件):

```
import tkinter as tk
import tkinter.messagebox as mb
import oxo_logic

top = tk.Tk()

def buildMenu(parent):
    menus = (
        ("File", (("New", evNew),
                  ("Resume", evResume),
                  ("Save", evSave),
                  ("Exit", evExit))),
        ("Help", (("Help", evHelp),
                  ("About", evAbout)))
    )
    menubar = tk.Menu(parent)
    for menu in menus:
        m = tk.Menu(parent)
        for item in menu[1]:
            m.add_command(label=item[0], command=item[1])
        menubar.add_cascade(label=menu[0], menu=m)

    return menubar

def dummy():
    mb.showinfo("Dummy", "Event to be done")

evNew = dummy
evResume = dummy
evSave = dummy
evExit = top.quit
evHelp = dummy
evAbout = dummy
```



```
mbar = buildMenu(top)
```

```
top["menu"] = mbar
```

```
tk.mainloop()
```

在初始导入和创建顶层小部件之后，定义了创建菜单的函数。

菜单结构被定义为一组嵌套元组。叶节点菜单项包含了名称-函数对。然后创建顶层菜单栏对象并循环数据结构，创建了子菜单并将它们插入到菜单栏中，最后返回完整的菜单栏对象。

下一部分代码定义了事件处理函数——至少，它在你完成时会这么做。现在，你只是简单地定义了一个虚拟函数(dummy function)来处理所有的事件(除了不重要的 `evExit`)并把它赋值给每个事件处理变量。很快，你会回到这部分并把它们转换为游戏的真实事件处理函数的定义。

最后，执行 `buildMenu()` 函数并把结果赋值给 `top` 小部件的 `menu` 属性。然后执行 `mainloop()`。

当运行这个程序时，应该会得到一个带有菜单栏的窗口，并且当单击任何菜单项时它都会调用虚拟函数。

4.8.3 创建 Tic-Tac-Toe 面板

在创建菜单结构之后，现在想要扩展程序来创建面板。该面板位于一个 `Frame` 中，而这个 `Frame` 本身也位于另一个外层 `Frame` 的正中央。这样设计的理由是：可以将为面板按钮布局的任务和面板整体布局的任务拆开来。

与创建菜单的方式相同，你会创建一个负责构建面板的函数。面板是游戏的一部分，与逻辑层和基本的游戏玩法相连接。所以也需要定义代码将游戏的逻辑层数据模型转换为 GUI 中显示的面板，反之亦然。可以在小的辅助函数中做这些操作。还需要添加一个事件处理函数。在按钮被单击时，它会设置按钮标签。GUI 和逻辑层之间的相互作用也是按钮单击事件处理程序代码的一部分。

先处理 GUI 创建部分。代码如下(可以从 `zip` 文件的 `OXO` 文件夹中加载 `oxo_gui_board.py`)：

```
def evClick(row,col):
    mb.showinfo("Cell clicked", "row:{}, col:{}".format(row,col))

def buildBoard(parent):
    outer = tk.Frame(parent, border=2, relief="sunken")
    inner = tk.Frame(outer)
    inner.pack()

    for row in range(3):
        for col in range(3):
            cell = tk.Button(inner, text=" ", width="5", height="2",
```

```

        command=lambda r=row, c=col : evClick(r,c) )
    cell.grid(row=row, column=col)
    return outer

mbar = buildMenu(top)
top["menu"] = mbar

board = buildBoard(top)
board.pack()
status = tk.Label(top, text="testing", border=0,
                  background="lightgrey", foreground="red")
status.pack(anchor="s", fill="x", expand=True)

tk.mainloop()

```

已经开始使用 Tkinter 的一些更加美观的特性来改善小部件的外观,并且更加清晰地在屏幕上分隔它们。本例中使用了 `border` 和 `relief` 属性来让面板相比于菜单和状态栏显得更加界限清晰(也定义了状态栏,因为它只是两行额外的线,并且可以让你更加接近最终的 GUI 结构)。本例中还为状态栏设置了颜色选项并把它锚定在顶层框架的底部(通过使用 `s` 值(也就是 `south`)来指定)。

面板构造本身只是几个创建网格模式的 `for` 循环(宽和高的值是通过反复试验确定的)。按钮的 `command` 参数必须是一个不接受任何参数的函数,因为它使用 `lambda` 函数机制,但是需要输入行和列的值,为此设置了两个有默认值的参数。在本示例中,行和列的值是按钮创建的点坐标。每个按钮使用它自己独特的一组值来调用 `evClick` 函数。这是使用 Tkinter 编程的一种常用技巧。你也使用了网格布局管理器而不是封装器,因为面板布局完美地匹配网格布局风格。只需要指定每个控件的行和列的位置,网格就会完成余下的工作。

4.8.4 将 GUI 连接到游戏

在完成基本的 GUI 结构之后,现在就可以将注意力转移到编写游戏代码并将各种菜单绑定到最终的事件处理函数上。可以首先处理游戏代码,因为它主要是在 `evClick` 事件处理程序中,并且可以从几个辅助函数获得帮助,可以调用 `cells2game` 和 `game2cells`。

对代码的修改如下(可以从 `zip` 文件的 `OXO` 文件夹中加载 `oxo_gui_game.py`):

```

gameover = False
def evClick(row,col):
    global gameover
    if gameover:
        mb.showerror("Game over", "Game over!")
        return
    game = cells2game()
    index = (3*row) + col
    result = oxo_logic.userMove(game, index)
    game2cells(game)

    if not result:

```

```

        result = oxo_logic.computerMove(game)
        game2cells(game)
    if result == "D":
        mb.showinfo("Result", "It's a Draw!")
        gameover = True
    else:
        if result == "X" or result == "O":
            mb.showinfo("Result", "The winner is: {}".format(result))
            gameover = True

def game2cells(game):
    table = board.pack_slaves()[0]
    for row in range(3):
        for col in range(3):
            table.grid_slaves(row=row, column=col)[0]['text'] =
                game[3*row+col]

def cells2game():
    values = []
    table = board.pack_slaves()[0]
    for row in range(3):
        for col in range(3):
            values.append(table.grid_slaves(row=row,
                column=col)[0]['text'])
    return values

```

如果对比 `evClick` 代码和原始的 `playGame()` 函数，你会发现有很多相似之处。`game2cells()` 函数类似于原始的 `printGame()` 函数。`cells2game()` 函数使用了一些小部件方法来获取小部件，在这里是按钮。作为备选方案，也可以把按钮列表存储在一个全局的数据结构中，这样可以更加直接地访问它们。尽管用户界面发生了很大改变，但是 `userMove` 和 `computerMove` 的游戏逻辑并没有改变。

最后，填充菜单事件处理程序。这些就不是很重要了，而最终的程序如下(可以从 zip 文件的 OXO 文件夹中加载 `oxo_gui_complete.py`)：

```

import tkinter as tk
import tkinter.messagebox as mb
import oxo_logic

top = tk.Tk()

def buildMenu(parent):
    menus = (
        ("File", (
            ("New", evNew),
            ("Resume", evResume),
            ("Save", evSave),
            ("Exit", evExit))),
        ("Help", (
            ("Help", evHelp),
            ("About", evAbout)))
    )

```



```

    )

    menubar = tk.Menu(parent)
    for menu in menus:
        m = tk.Menu(parent)
        for item in menu[1]:
            m.add_command(label=item[0], command=item[1])
        menubar.add_cascade(label=menu[0], menu=m)

    return menubar

def evNew():
    status['text'] = "Playing game"
    game2cells(oxo_logic.newGame())

def evResume():
    status['text'] = "Playing game"
    game = oxo_logic.restoreGame()
    game2cells(game)

def evSave():
    game = cells2game()
    oxo_logic.saveGame(game)

def evExit():
    if status['text'] == "Playing game":
        if mb.askyesno("Quitting", "Do you want to save the game before
            quitting?"):
            evSave()
    top.quit()

def evHelp():
    mb.showinfo("Help", '''
    File->New: starts a new game of tic-tac-toe
    File->Resume: restores the last saved game and commences play
    File->Save: Saves current game.
    File->Exit: quits, prompts to save active game
    Help->Help: shows this page
    Help->About: Shows information about the program and author''')

def evAbout():
    mb.showinfo("About", "Tic-tac-toe game GUI demo by Alan Gauld")

def evClick(row, col):
    if status['text'] == "Game over":
        mb.showerror("Game over", "Game over!")
        return

    game = cells2game()
    index = (3*row) + col

```

```

result = oxo_logic.userMove(game, index)
game2cells(game)

if not result:
    result = oxo_logic.computerMove(game)
    game2cells(game)
if result == "D":
    mb.showinfo("Result", "It's a Draw!")
    status['text'] = "Game over"
else:
    if result == "X" or result == "O":
        mb.showinfo("Result", "The winner is: {}".format(result))
        status['text'] = "Game over"

def game2cells(game):
    table = board.pack_slaves()[0]
    for row in range(3):
        for col in range(3):
            table.grid_slaves(row=row, column=col)[0]['text'] = game[3*row+col]

def cells2game():
    values = []
    table = board.pack_slaves()[0]
    for row in range(3):
        for col in range(3):
            values.append(table.grid_slaves(row=row,
            column=col)[0]['text'])
    return values

def buildBoard(parent):
    outer = tk.Frame(parent, border=2, relief="sunken")
    inner = tk.Frame(outer)
    inner.pack()
    for row in range(3):
        for col in range(3):
            cell = tk.Button(inner, text=" ", width="5", height="2",
            command=lambda r=row, c=col: evClick(r,c) )
            cell.grid(row=row, column=col)
    return outer

mbar = buildMenu(top)
top["menu"] = mbar

board = buildBoard(top)
board.pack()
status = tk.Label(top, text="Playing game", border=0,
    background="lightgrey", foreground="red")
status.pack(anchor="s", fill="x", expand=True)

```

```
tk.mainloop()
```

事件函数基本上与原始函数相同，因为它们只是调用 `oxo_logic` 函数，然后使用 `game2shell()` 函数来显示面板。现在使用状态文本而不是全局的 `gameover` 标志。Help 菜单仅在 `showinfo` 对话框中显示文本。

最终的游戏如图 4-6 所示。

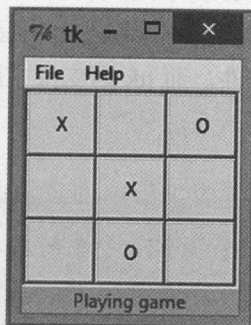


图 4-6 最终的 Tkinter GUI

还可以做很多工作来润色这个游戏，但是它现在已经足够演示 Tkinter 作为用户界面工具包的出色能力了，同时也再一次证明了逻辑层和数据层与展示层分开的威力。你现在已经编写超过 600 行代码来试验各种版本的 tic-tac-toe。这对任何人来说都已足够，所以是时候接触新内容了。

Tkinter 有许多其他的小部件和技巧。我们首先试验简单的 GUI 应用，然后添加或修改不同的选项来影响布局和外观。你甚至还不了解如何显示格式化的文本、图片、画图或创建复杂对话框。所有这些都是可以在现有基础上实现的。我们身边有很多 Tkinter 教程和示例程序，包括 Python 的默认 IDE (IDLE)。可以通过阅读代码来了解这些程序是如何控制外观和使用小部件的，这是一种很好的学习方式。

注意：在 New Mexico Tech 上有一个非常好的关于 Tkinter 编程的在线教程。它包括一些关于 Tkinter 较新特性的资料：<http://infohost.nmt.edu/tcc/help/pubs/lang.html>。

遗憾的是，它是基于旧版本 Python2 Tkinter 包结构的。所以在版本 3 上需要做一些微调。

还有一本 John Grayson 编写的专注于 Tkinter 的书：*Python and Tkinter Programming*。尽管现在看来有些过时，但确实包含了一些有用的引用资料和一些比你在其他教程中找到的更长、也更加复杂的示例。

Mark Lutz 也在他的参考书籍 *Programming Python* (O'Reilly, 2011) 中包含了一些关于 Tkinter 的内容。Lutz 这本书的最新版本使用了 Python3。

标准库中还有一些 Tkinter 的扩展模块(尽管一些 Linux 发布版本因为某些原因删除了它们)。在下一节中, 会看到这些扩展模块可以为程序添加哪些功能。

4.9 扩展 Tkinter

Tkinter 的两个最大问题是没有足够的小部件并且外观很丑。相比于其他 GUI 工具, 这些问题确实存在。然而, 在最近的更新版本中, Tkinter 引入了两个新模块 `tix` 和 `ttk` 来改善这些问题。`tix` 添加了几个新的小部件, 而 `ttk` 则支持主题。从本质上说, GUI 拥有更接近于原生操作系统的 GUI。



注意: `tix` 和 `ttk` 都依赖于 Tcl/Tk 包中的底层库。如果在使用 `tix` 和 `ttk` 时遇到了困难, 请检查是否安装了 Tcl/Tk 库。通常, Python 安装程序会为你做这些事情, 但有时可能会产生冲突, 需要手动解决。

遗憾的是, 这些模块的文档并不像你想象的那样全面。在本书成文时, Python 文档只包含了 Tcl/Tk 文档的链接。在熟悉之后, 通常可以知道所需的选项, 但这并不是最理想的。然而, Python 交互式提示符可以提供帮助。可以用它来试验这些模块并熟悉它们的功能。这两个模块支持大部分之前使用过的 Tkinter 小部件, 所以应该能够非常轻松地将 Tkinter 程序转向 `ttk` 或 `Tix`。对于 `Tix`, 修改 `tkinter` 的导入行:

```
import tkinter.tix as tk
```

在导入时使用 `tk` 别名的好处是: 不必将 `tkinter` 前缀改为 `tix`。你只是简单地导入新模块并使用一样的别名。如果喜欢的话, 可以在 `tic-tac-toe` 游戏上试试。它与 Tkinter 的作用方式相同, 只是窗口的标题栏显示的是 `tix` 而不是 `Tk`。

`ttk` 要稍微复杂些。因为 `ttk` 使用 `tkinter` `mainloop` 和顶层窗口, 所以需要导入它们。然后, 在创建小部件时会引用 `ttk`, 而在控制顶层窗口和事件循环时会使用 `tkinter`。在之后的 `ttk` 练习中会介绍它们。

4.9.1 使用 tix

由于 `tix` 与 `tkinter` 十分相似, 因此可以将所有关于 `tkinter` 的知识转换到 `tix`, 然后直接学习新的小部件。它包含 40 多个新的小部件, 但是有一些没有很好地文档化, 即使是在 Tcl/Tk 社区中。如果紧盯 Python 文档页的子集, 就不会有什么问题了。在这里, 你只会浅尝辄止, 但是希望这已足够证明 `tix` 对于 Tkinter 家族是一个很有价值的补充。

在此你将了解的小部件包括 `ComboBox`、`ScrolledText` 和 `Notebook`。

通过如下代码, 可以了解如何使用 `ComboBox` 输入控件来设置标签文本:

```

>>> import tkinter.tix as tix
>>> top = tix.Tk()
>>> lab = tix.Label(top)
>>> lab.pack()
>>> cb = tix.ComboBox(top,command=lambda s:lab.config(text=s))
>>> for s in ["Fred","Ginger","Gene","Debbie"]:
...     cb.insert("end",s)
...
>>> cb.pick(0)
>>> lab['text'] = "Pick any item"
>>> cb.pack()
>>> top.mainloop()
>>>

```

有几件事情需要注意。首先,使用的是 `config()` 而不是之前使用的字典来设置文本属性。`config()` 的优势在于可以通过把它们作为命名的参数来同时设置多个属性。其次,事件处理程序 `lambda` 函数使用了字符串参数 `s`。它是通过小部件事件传入的。字符串保存当前选择的值。图 4-7 显示了结果窗口。

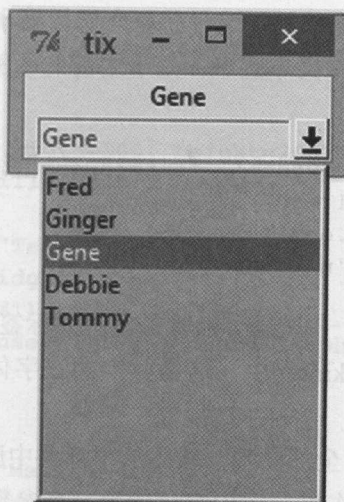


图 4-7 一个 tix 组合框

`ScrolledText` 小部件是标准 `Text` 小部件的扩展,它可以显示图片以及格式化的文本。`tix` 版本的 `ScrolledText` 小部件自动添加了滚动条,这非常有用,因为使用标准工具包来实现这一功能会产生很大的工作量。在使用方法上,它与其他 Tkinter 小部件非常相似。可以输入下面的代码试一试:

```

>>> top = tix.Tk()
>>> st = tix.ScrolledText(top, width=300, height=100)
>>> st.pack(side='left')
>>> top.mainloop()

```

图 4-8 显示了结果文本框,文本量足够激活垂直滚动条。

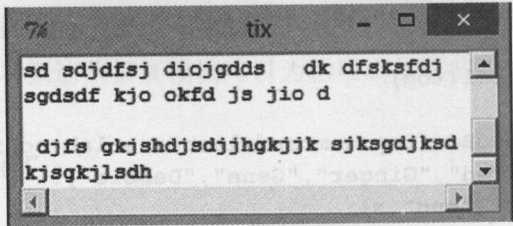


图 4-8 运行中的 tix ScrolledText 小部件

可以手动在文本框中输入或以编程的方式插入文本，如下：

```
>>> t = st.subwidget('text')
>>> t
<tkinter.tix._dummyText object at 0x019186D0>
>>> t.insert('0.0',"Some inserted text")
>>> t.insert('end',"\\n more inserted text")
>>>
```


注意，使用了 `subwidget()` 方法来获得底层文本小部件的一个引用，然后使用它的 `insert()` 方法插入文本。在使用 Tix 小部件时，这种获取底层标准小部件的技术是非常常用的。

也可以在文本小部件中选中文本区域。基于之前的示例，可以修改首行文本的字体和大小，如下所示：

```
>>> t.tag_configure('newfont', font=("Roman", 16, "bold"))
>>> s = t.get('1.0','1.end')
>>> t.delete('1.0','1.end')
>>> t.insert('1.0',s,'newfont')
```

`tag_configure()` 方法新建了一个 tag，也就是文本的标签。标签被称为 `newfont` 并使用加粗的 16 号 Roman 字体。在 Tkinter 中，这是标准的三字体说明符格式(因此 Tix 和 ttk 也支持)。

然后使用 `get()` 获得首行的全部文本，然后从小部件中删除了当前文本，并把它替换为相同文本但使用 `newfont` 标签作为 `insert()` 方法的第三个参数。



注意：Tkinter 中的文本索引使用包含格式化成浮点数的字符串。但实际上，它包含了用点分隔的行和列坐标。行开始于 1，但列开始于 0。所以 1.0 是第一行的第一个字符。可以使用字符串 `end` 或预定义的常量 `tkinter.END` 来标识行的结束，或者所有文本的结束。具体情况由语意来决定。

结果如图 4-9 所示。注意，虽然所有文本都显示在 Tix 的 `scrolledText` 小部件中，但是实际上使用了底层的标准 Tkinter `Text` 小部件来处理文本。

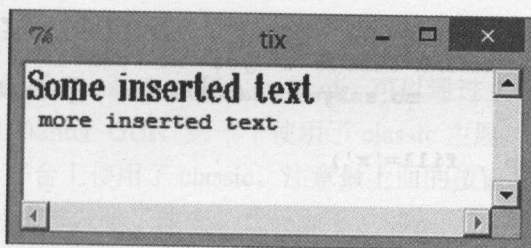


图 4-9 修改文本小部件中的文本外观

最后一个要熟悉的小部件是 NoteBook 小部件。它更加复杂，包含一些页。每页都有一个相关的标签。用户可以选择标签来激活相关的页。页只是一个 Tkinter 容器。可以使用任何控件来填充它。通常，它是一个文本窗口或窗体。将会创建一个两页的 notebook。第一页包含一个 ScrolledText 小部件，而另一页包含一组用于启动各种消息对话框的按钮。

为了实践一下 NoteBook，将下面的代码输入文件中，并从命令行或 IDE 中执行它(或者如果你愿意，可以从 zip 文件的 Tkinter 文件夹中加载 tix-notebook.py 文件)：

```
import tkinter.tix as tix
import tkinter.messagebox as mb

top = tix.Tk()

nb = tix.NoteBook(top, width=300, height=200)
nb.pack(expand=True, fill='both')

nb.add('page1', label="Text")
f1 = tix.Frame(nb.subwidget('page1'))
st = tix.ScrolledText(f1)
st.subwidget('text').insert("1.0", "Here is where the text goes...")
st.pack(expand=True)
f1.pack()

nb.add('page2', label="Message Boxes")
f2 = tix.Frame(nb.subwidget('page2'))
tix.Button(f2, text="error", bg="lightblue",
            command=lambda t="error", m="This is bad!":
                mb.showerror(t,m) ).pack(fill='x',expand=True)
tix.Button(f2, text="info", bg='pink',
            command=lambda t="info", m="Information":
                mb.showinfo(t,m) ).pack(fill='x',expand=True)
tix.Button(f2, text="warning", bg='yellow',
            command=lambda t="warning", m="Don't do it!":
                mb.showwarning(t,m) ).pack(fill='x',expand=True)
tix.Button(f2, text="question", bg='green',
            command=lambda t="question", m="Will I?":
                mb.askquestion(t,m) ).pack(fill='x',expand=True)
tix.Button(f2, text="yes - no", bg='lightgrey',
            command=lambda t="yes - no", m="Are you sure?":
                mb.askyesno(t,m) ).pack(fill='x',expand=True)
```

```
tix.Button(f2, text="yes - no - cancel", bg='black', fg='white',
          command=lambda t="yes - no - cancel", m="Last chance...":
            mb.asksyesnocancel(t,m) ).pack(fill='x',expand=True)

f2.pack(side='top', fill='x')

top.mainloop()
```

在这个示例中，导入了模块并同样创建了顶层小部件。然后，创建了一个名为 nb 的 `tix.Notebook` 对象，并为这个对象添加了 `page1` 页。然后创建了一个窗体并把它父类指向刚刚创建的 `page1` 页。之后添加了一个文本小部件和一些文本，并封装了这个小部件和窗体。

接下来，创建了第二页 `page2`，并像之前一样为此页添加了一个窗体。接着创建了一组按钮，并使用 `lambda` 函数作为命令将这些按钮链接到各种消息框上。修改了按钮和窗体的 `pack()` 选项，让按钮占据页的全部宽度，并且使用了不同的颜色来突显它们。

注意，对于 `notebook`、按钮和 `SrolledText` 小部件，我们为封装器指定了 `expand` 选项。`expand` 告诉布局管理器在窗口改变大小时扩展该控件。通过联合使用 `expand`、`fill` 和 `anchor`，在窗口大小改变时，可以精确地控制小部件的行为(建议你试试这些选项)。

最后，启动了 `mainloop()` 函数。当执行该函数时，结果如图 4-10 所示，它显示了运行中的两页 `notebook`。

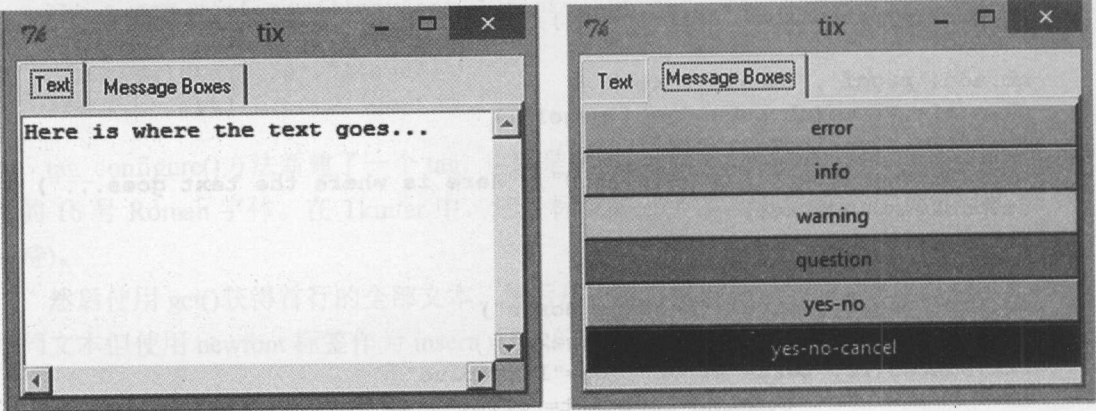


图 4-10 显示两页的 Tix Notebook

4.9.2 使用 ttk

如前所述，`ttk` 模块为 `Tkinter` 和 `Python` 添加了主题的概念。主题是一种图形风格，它可以让同样的 GUI 结构拥有与原生操作系统相同的外观和感觉。虽然 `ttk` 自带了几个主题，但是也可以创建自己的主题。

预定义的主题会随着操作系统而发生变化。通过查看 `ttk.Style().theme_names()` 的输出，可以找到自己平台的主题名。在 `Windows` 上，它们包括 `Classic`(默认)、`winnative`、`vista` 和 `xpnative`。

ttk 自带了 11 个标准 Tkinter 小部件，可以用主题改变它们的外观。另外，ttk 还包含 6 个它自己的新小部件，包括 ComboBox 和 NoteBook。可以通过主题来改变应用的外观。图 4-11 展示了一个简单的 Tkinter GUI。第一个使用了 classic 主题，第二个使用了 vista，而第三个在 Ubuntu Linux 平台上使用了 classic。注意最上面的按钮并没有很大的变化，但是新的 ttk 按钮在每张图片中都发生了改变。代码如下：

```
>>> import tkinter as tk
>>> import tkinter.ttk as ttk
>>> top = tk.Tk()
>>> s = ttk.Style()
>>> s.theme_use('classic')
>>> tk.Button(top, text="old button").pack()
>>> ttk.Button(top, text="new button").pack()
>>>
```

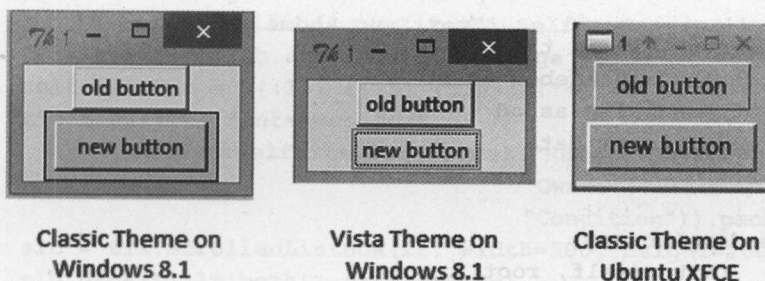


图 4-11 各种各样的 ttk 主题

很明显，需要将样式对象中的 classic 换成 vista 来获得 Vista 主题。需要小量额外的工作来定义样式对象。但除此之外，使用 ttk 基本上与普通 Tkinter 编程一样。当运行代码时，应该注意到按钮发生的改变不仅仅是外观上的简单调整。例如，当鼠标移到按钮上时，它的颜色会发生变化。

现在真正需要了解的就是这些 ttk 内容。还有很多可以尝试的选项，甚至可以定制自己的样式和主题。在大多数情况下，只是像上面那样使用它，而 Tkinter 程序的外观和感觉会有很小但很重要的改善。

4.10 再次回顾借出库

现在你已经熟悉了 Tkinter 工具包的几个部分。是时候在大一点的示例中将这部分合在一起了，这个示例应该能够比 tic-tac-toe 游戏实现更多的功能。你将使用在第 3 章创建的借出库数据库，并创建一个 GUI 前端。这将有助于你巩固已学的知识，但也会引入一些新元素和技术：

- ScrolledListBox 小部件
- 如何捕捉底层事件，比如双击鼠标和窗口级别的事件
- 如何创建和使用自定义对话框

- 如何设置字体
- 如何使用 state 属性激活/停用小部件
- 如何使用面向对象的技术创建 GUI

你将在接下来的“试一试”中创建 GUI。

试一试：创建借出库 GUI

在这个“试一试”中，将为在第 3 章创建的借出库数据库添加一个 GUI(由于有很多代码，所以你可能想要从第 4 章的 zip 下载文件中加载它)。完成下面这些步骤：

(1) 创建项目文件夹 Lendy，然后将第 3 章的 lendydata.py 模块以及 lendy.db 数据库复制到新文件夹中(也可以使用 Chapter 2 项目文件夹，这由你决定)。

(2) 打开文本编辑器或 IDE，创建 lendy-gui.py 文件(或者从 zip 文件中加载它)。输入下面的代码：

```
import tkinter.tix as tix
import tkinter.messagebox as mb
import optionsdialog as od
import lendydata as data
import os

class LendingLibrary:
    def __init__(self, root):
        self.isDirty = False
        self.top = root
        root['menu'] = self.buildMenus(root)
        mainWin = self.buildNoteBook(root)
        mainWin.pack(fill='both', expand=True)
        self.top.protocol('WM_DELETE_WINDOW', self.evClose)
        self.top.title('Lending Library')
        data.initDB() # use default file
        self.items = data.get_items()
        self.members = data.get_members()
        self.populateItemList()
        self.populateMemberList()

    def buildMenus(self, top):
        menus= (
            ("Item", ((("New", self.evNewItem),
                        ("Edit", self.evEditItem),
                        ("Delete", self.evDeleteItem),
                        )),
            ("Member", ((("New", self.evNewMember),
                        ("Edit", self.evEditMember),
                        ("Delete", self.evDeleteMember),
                        )),
            ("Help", ((("Help", self.evHelp),
                        ("About", lambda : mb.showinfo(
                            "Help About",
```

```

        "Lender application\nAuthor: Alan Gauld""))
    ))))

    self.menubar = tix.Menu(top)
    for menu in menus:
        m = tix.Menu(top)
        for item in menu[1]:
            m.add_command(label=item[0], command=item[1])
        self.menubar.add_cascade(label=menu[0], menu=m)
    return self.menubar

def buildNoteBook(self, top):
    mono_font = self.getMonoFont()
    nb = tix.NoteBook(top)

    nb.add("itemPage", label="Items",
           raisecmd=lambda pg="item": self.evPage(pg))
    fr = tix.Frame(nb.subwidget("itemPage"))
    self.itemFmt = "{:15} {:15} {:10} ${:<8} {:12}"
    tix.Label(fr, font=mono_font,
              text=self.itemFmt.format("Name", "Description",
                                       "Owner", "Price",
                                       "Condition")).pack(anchor='w')
    slb = tix.ScrolledListBox(fr, width=500, height=200)
    slb.pack(fill='both', expand=True)
    fr.pack(fill='both', expand=True)
    self.itemList = slb.subwidget("listbox")
    self.itemList.configure(font=mono_font, bg='white')
    self.itemList.bind('<Double-1>', self.evEditItem)

    nb.add("memberPage", label="Members",
           raisecmd=lambda pg="member": self.evPage(pg))
    fr = tix.Frame(nb.subwidget("memberPage"))
    self.memberFmt = "{:<15} {:<40}"
    tix.Label(fr, font=mono_font,
              text=self.memberFmt.format("Name", "Email Address")).pack
              (anchor='w')
    slb = tix.ScrolledListBox(fr, width=40, height=20)
    slb.pack(fill='both', expand=True)
    fr.pack(fill='both', expand=True)
    self.memberList = slb.subwidget("listbox")
    self.memberList.configure(font=mono_font, bg='white')
    self.memberList.bind('<Double-1>', self.evEditMember)

    return nb

def getMonoFont(self):
    if os.name == 'nt':
        return ('courier', '10', '')
    else:

```

```

        return ('mono', '10', '')

    def populateItemList(self):
        self.itemList.delete('0', 'end')
        for item in self.items:
            item = list(item[1:])
            item[2] = data.get_member_name(item[2])
            self.itemList.insert('end', self.itemFmt.format(*item))

    def populateMemberList(self):
        self.memberList.delete('0', 'end')
        for mbr in self.members:
            self.memberList.insert('end', self.memberFmt.format(*mbr[1:]))

    def evClose(self, event=None):
        data.closeDB()
        self.top.quit()

```

(3) 如果手动输入代码，你可能想要停下来，将所有创建并绑定小部件的代码行中指向事件处理程序的引用(self.evXXXXX)，除了 evClose，都替换为 None。这能让你执行 GUI 并观察它的外观。在执行代码前，需要在文件底部添加下面的启动代码：

```

if __name__ == "__main__":
    top = tix.Tk()
    app = LendingLibrary(top)
    top.mainloop()

```

(4) 查看了运行中的 UI 后，现在可以恢复类代码中原始的事件处理程序引用，然后继续向类定义中添加以下事件处理程序代码：

```

##### notebook event handler #####
def evPage(self, page):
    if page=='item':
        self.menubar.entryconfigure('Item', state='active')
        self.menubar.entryconfigure('Member', state='disabled')
    if page=='member':
        self.menubar.entryconfigure('Item', state='disabled')
        self.menubar.entryconfigure('Member', state='active')

##### Item Event Handlers #####
def evNewItem(self):
    dlg = od.OptionsDialog(top, (
        ["Name", "" ],
        ["Description", "" ],
        ["Owner", "" ],
        ["Price", "" ],
        ["Condition", "" ]))

    if dlg.changed:
        ownerID = self.get_member_id(dlg.options[2][1])
        data.insert_item(dlg.options[0][1],dlg.options[1][1],

```



```

        ownerID, int(dlg.options[3][1]),
        dlg.options[4][1])
    self.items = data.get_items()
    self.populateItemList()

def evEditItem(self, event=None):
    # get selected member
    indices = self.itemList.curselection()
    index = int(indices[0]) if indices else 0
    item = self.items[index]
    ownerID = item[3]
    ownerName = data.get_member_name(ownerID)
    dlg = od.OptionsDialog(top, (
        ["Name", item[1] ],
        ["Description", item[2] ],
        ["Owner", ownerName ],
        ["Price", item[4] ],
        ["Condition", item[5] ]))
    if dlg.changed:
        if dlg.options[2][1] != ownerName: # its changed
            ownerID = self.get_member_id(dlg.options[2][1])
            data.update_item(item[0],dlg.options[0][1],dlg.options[1][1],
                            ownerID, int(dlg.options[3][1]),
                            dlg.options[4][1])
            self.items = data.get_items()
            self.populateItemList()

def evDeleteItem(self):
    indices = self.itemList.curselection()
    index = int(indices[0]) if indices else 0
    item = self.items[index]
    data.delete_item(item[0])
    self.items = data.get_items()
    self.populateItemList()

# Ideally should use a combo box in options dialog.
# this gives potential error if more than one member with same name
def get_member_id(self, name):
    for member in self.members:
        if member[1] == name:
            return member[0]

##### Member Event Handlers #####
def evNewMember(self):
    dlg = od.OptionsDialog(top, (
        ["Name", ""],
        ["Email", ""]))
    if dlg.changed:
        data.update_member(None,dlg.options[0][1],dlg.options[1][1])
        self.members = data.get_members()

```

```

        self.populateMemberList()

def evEditMember(self, event=None):
    indices = self.memberList.curselection()
    index = int(indices[0]) if indices else 0
    mbr = self.members[index]
    dlg = od.OptionsDialog(top, (
        ["Name", mbr[1]],
        ["Email", mbr[2]]))

    if dlg.changed:
        data.update_member(mbr[0], dlg.options[0][1], dlg.options[1][1])
        self.members = data.get_members()
        self.populateMemberList()

def evDeleteMember(self):
    indices = self.memberList.curselection()
    index = int(indices[0]) if indices else 0
    mbr = self.members[index]
    data.delete_member(mbr[0])
    self.members = data.get_members()
    self.populateMemberList()
#### Help event handler #
def evHelp(self):
    mb.showinfo("Help", "")

Lending Library Application

Item->New:
    Create a new item in the library
Item->Edit:
    Modify the attributes of the
    selected item (default is first)
Item->Delete:
    Delete selected item (no default)

Member->New:
    Add a member to the library
Member->Edit:
    Modify selected members data
    (default is first)
Member->Delete:
    Delete selected member (no default)

Help->Help:
    Display this screen
Help->About:
    About the program.""")

if __name__ == "__main__":
    top = tix.Tk()
    app = LendingLibrary(top)

```

```
top.mainloop()
```

(5) 代码使用了另一个名为 OptionsDialog 的类，该类在另一个文件中定义。新建一个名为 optionsdialog.py 的文件(或者从 Chapter4.zip 文件中打开它)。这个文件包含以下代码：

```
import tkinter.tix as tix
import tkinter.simpledialog

class OptionsDialog(tkinter.simpledialog.Dialog):
    def __init__(self, master, options, *args):
        self.options = options
        self.entries = []
        self.changed = False
        super().__init__(master, *args)

    def body(self, top):
        ''' define GUI elements'''
        f = tix.Frame(top)
        f.pack(expand=True, fill='x')
        for row, opt in enumerate(self.options):
            tix.Label(f, text=opt[0]).grid(row=row, column=0, sticky='w')
            e = tix.Entry(f)
            e.grid(row=row, column=1, sticky='e')
            e.insert('end', str(opt[1]))
            self.entries.append(e)

    def apply(self):
        ''' store entry values in options '''
        for index, opt in enumerate(self.options):
            opt[1] = self.entries[index].get()
        self.changed = True

if __name__ == "__main__":
    top = tix.Tk()
    app = OptionsDialog(top, (["First", "my value"], ["Second", "Another
value"]))
    top.mainloop()
```

(6) 运行 lendy-gui.py。

你应该发现物品和成员被列在它们各自的标签下，而菜单会与标签同步激活或停用。可以使用菜单来编辑标签行，也可以通过双击来实现。窗口关闭图标应该干净利落地关闭所有东西。

示例说明

主 GUI 使用了与之前 GUI 应用相同的基本模式。但是这一次把代码放入了一个类中。类初始化器 __init__() 设置了一些实例变量并调用了各种辅助函数来创建 GUI。第一个辅助函数从数据结构中创建了菜单，就像在 tic-tac-toe GUI 中所做的那样。第二个辅助函数创

建了 `tix.NoteBook`，它也是应用的主窗口。一个很大的区别使用的是 `tix.ScrolledListbox` 而不是 `tix.ScrolledText` 小部件。使用 `Scrolled ListBox` 小部件可以更加容易地选择行。为了帮助布局，尤其是在 `NoteBook` 页顶部将列与 `Label` 栏对齐，将字体修改为等宽。为了克服安装的字体在不同操作系统上的差异问题，根据在第 2 章学到的 `os.name` 属性将字体设置为一个适当的值。还将背景色设置为白色，这有助于突显被选中的行。

使用 `bind()` 方法将鼠标左键双击(<Double-1>)链接到适当的编辑事件处理程序上。除了处理默认小部件的命令，`bind()` 机制可以处理任意事件。不同的是，回调函数必须接受一个事件参数。事件对象将包含诸如按了哪个键、鼠标位置或任何处理事件所需要的信息。在本示例中，只是忽略了事件对象并为参数设置了默认值 `None`。这样普通的回调机制以及双击链接也可以使用它。使用了 `lendydata` 模块中的转换函数 `get_member_name()` 将数据库返回的 `OwnerID` 值转换成了在显示时更加方便用户理解的名称。但是，没有为给定名称返回 `OwnerID` 的反转函数，因为可能有多个成员拥有相同的名字。为了应对这种情况，创建了自己的辅助函数 `get_member_id()`。它仅返回 `self.items` 中第一个匹配的 `ID` (理想状态下，可能会在选项对话框中创建一个组合框，但是这会使对话框的数据驱动变得非常复杂。现在，需要做一些妥协)。

事件处理程序都是自解释型的。在选中新的 `NoteBook` 页时，就会触发 `evPage()` 处理程序。它只是为活动页面启动对应的菜单。这保证你在查看物品时不能使用成员函数，反之亦然。在此只是简单地修改相关 `Menu` 小部件的 `state` 属性。

`new` 和 `edit` 事件处理程序包含很多类似的代码。应该创建一个辅助函数来简化它们并避免重复的工作。但是这样做会导致安排正确输入值的额外复杂度，所以我们决定保持不变。这两个事件处理程序都使用 `get_member_id()` 方法从被提供的拥有者名字值中获得 `OwnerID` 值。

同样的，本应该有一个辅助函数用于选出被选中的下标。但是由于函数只有两行代码，所以我们仍然让它保持不变。注意，选择可能是下标的列表。而这些会作为字符串返回。所以需要显式地将字符串转换为数字下标。

如你所愿，大部分工作都是在数据模块中完成的。毕竟，GUI 应该只负责显示。你可能注意到这个应用没有核心逻辑层，那是因为你本质上只是在修改数据，没有涉及业务逻辑，所以不需要核心逻辑层。GUI 只是直接调用数据层。通过把数据层暴露为一个 API，允许在没有任何 SQL 引用的情况下编写 GUI。这意味着可以将底层的 `SQLite` 实现替换为 `Postgres` 或 `MySQL` 等，而 API 仍然保持不变，GUI 页也不需要改变。

最后一部分是 `OptionsDialog`，它在一个单独的模块中实现，因为你可能会在其他项目中复用这个模块。它接受一个名称和值的列表作为参数，并把它们显示为一个窗体。然后可以编辑值并保存任何改变。对话框被创建为一个类并继承于标准库的通用对话框框架，这个框架提供了几个方法，通过重写这些方法可以改变对话框的功能。创建对话框时会调用 `body()` 方法，GUI 布局的定义也是在该方法中完成的。用户单击 `OK` 按钮时会调用 `apply()` 方法。这个方法会以任何调用程序需要的方式处理和存储数据。还有一些其他方法可以帮

助改变按钮布局和行为以及验证数据的输入。在本示例中，只是把传入的数据存储在一个列表中，然后在用户单击 OK 按钮时更新列表(在 `apply()`方法中)即可。在对话框关闭后，可以通过调用访问列表提取新的值。

你已经已经接触到足够多的 Tkinter 知识，了解了如何为应用逻辑和数据包装一个 GUI。虽然它可能不是最强大的工具包，但使用起来很快捷且容易上手，而且它基于 Python。在下一节中，你将了解到一些更加强大的、可用的第三方 GUI 工具。

4.11 探索其他 Python GUI 工具包

有许多其他 GUI 工具包可用，包括特定于 Windows、MacOS X 和 X Windows 的原生工具，以及更加通用的多平台工具。大多数这些 GUI 工具包都有 Python 可用的包装器层(wrapper layer)。尽管一些工具包需要基于对象的方法而其他工具(比如 Tkinter)也允许过程式编程风格，但是它们的核心思想和概念都与 Tkinter 相同。如果觉得 Tkinter 不好用或者你的主要兴趣领域是 GUI 开发，那么这些工具包可能会有助于你。在接下来的章节中，你会了解每个工具包的优点和缺点。对于每个平台无关的工具包，还会有一个“hello world”风格的示例小程序。如果想要运行这些程序，需要安装对应的工具包，因为这些工具包都是由第三方提供的。

俗语说得好，没有绝对的最好和最坏。每个工具包都有它的爱好者，而不同程序员喜欢不同的工具包。建议你花一些时间来试一试感兴趣的工具包。至少要看一遍它们的介绍教程，了解它们是否适合你的代码风格。另外，还有一些有用的在线视频也介绍它们的特性。

4.11.1 wxPython

这是一个已经存在很久的工具包，它是基于 C++ wxWidgets 项目的一个包装器。wxWidgets 是一个 C++工具包。它能工作在所有常用的操作系统上，并保持原生系统的外观和感觉。版本 3.0 的 wxPython 发布于 2013 年年末。

wxWidgets 工具包具有非常丰富的控件集和强大的特性，用于支持比如跨平台打印等功能(从 GUI 打印是很多听起来容易的功能之一，但其实很难实现!)一些图形化的 GUI 创建工具可以为你生成代码。也可以像在 Tkinter 中那样，通过用精致的代码来手工完成所有事情。尽管 wxPython 很强大，但是它仍然要比这里讨论的一些其他工具简单得多。

wxWidgets 和 wxPython 都有活跃的邮件列表和论坛。wxPython 上还有一些可以参阅的书籍，包括一本由首席开发者编写的书。wxPython 网站是 <http://www.wxpython.org>。

wxPython 示例程序如下：

```
import wx
```

```
app = wx.App(False)
frame = wx.Frame(None, wx.ID_ANY, size=(320,240), "Hello World")
frame.Show(True)

app.MainLoop()
```

4.11.2 PyQt

尽管 Qt 工具包一开始被开发为一款商业产品，但却在之后的 Linux 的 KDE 桌面环境中崭露头角。久而久之，Qt 的版权管理已经被简化了，现在它可以被广泛地应用在开源项目中，并支持大多数操作系统的原生外观和感觉。Qt 是一个 C++ 工具包，而 PyQt 是基于它的 Python 包装器。版本 5.2 在 2014 年年初发布。

一些数据可以展示 Qt 的规模：它有 400 多个类以及数千个函数和方法。学习曲线是相当陡峭，但是它的功能也很强大。一些高级特性只对商业用户（那些支付版权费的用户）开放，而这种免费和授权软件的混合模式可能是 Qt 以及 PyQt 的最大缺点。PyQt 拥有一个全功能的图形化 GUI 创建工具。

一种真正开源(LGPL)的可选方案已经以 PySide 的形式发布了。它提供与 PyQt 类似的功能。尽管 Nokia 拥有 Qt 工具包，但它还是开发了 PySide。PySide 的 1.2.1 版本发布于 2013 年年中。

对于 PyQt 编程，至少有两本可以参阅的书籍，网址为 <http://qt-project.org/wiki/PySide>。PyQt 示例程序如下：

```
import sys
from PyQt4 import QtGui

app = QtGui.QApplication(sys.argv)
win = QtGui.QWidget()
win.resize(320, 240)
win.setWindowTitle("Hello World!")
win.show()

sys.exit(app.exec_())
```

PySide 示例程序如下：

```
import sys
from PySide import QtGui

app = QtGui.QApplication(sys.argv)
win = QtGui.QWidget()
win.resize(320, 240)
win.setWindowTitle("Hello World!")
win.show()

sys.exit(app.exec_())
```


如你所见，除了导入语句，代码都是相同的。

4.11.3 PyGTK

Gimp ToolKit 或现在的 GTK+ 一开始是用 C 语言为 GNU GIMP 图形编辑器开发的 GUI 工具包。之后，它被开发为通用的图形化工具，并且已经成为许多 Linux 发布版本的 GNOME 桌面环境的开发工具包。PyGTK 是 GTK++ 工具包的 Python 包装器。然而，情况已经变得更加复杂了，现在已经有多个部分的 PyGTK 匹配各种各样的 GTK+ 部分。现在 PyGObject 是官方模块。它支持大部分的 GNOME 软件平台，包括 GUI。作为 GNU 的稳定部分，它是一个开源项目，所以它不需要应对复杂的授权问题，会定期发布新版本。

有一个称为 Glade 的图形化设计工具，可以用来创建 GUI。该工具属于典型的 GNU 风格，文档很全面但是并不适合初学者。Glade 系统非常强大，也支持多平台。一旦安装就非常易用。

有几本关于 GTK+ 编程的书籍，但是它们的重点在于底层的 C API 而不是 Python 绑定。一个很好的在线文档可以在 <https://live.gnome.org/PyGObject> 上找到。

GTK 示例程序如下：

```
from gi.repository import Gtk

win = Gtk.Window(title="Hello World")
win.resize(320,240)
win.connect("delete-event", Gtk.main_quit)
win.show_all()

Gtk.main()
```

4.11.4 原生 GUI: Cocoa 和 PyWin32

Cocoa 和 Win32 分别是 Mac OS X 和 Windows 操作系统的原生 GUI 工具包。它们都可以使用 Python 编程。Cocoa 的 PyObjC 工具包是由 MacPython 项目提供的，而原生的 MacOS X 开发工具可以用来创建 GUI 并将 GUI 连接到代码上。你已经在第 2 章接触过 Pywin32 包，其中讨论了它可以调用 Win32 API 的能力。Win32 API 不仅仅只是底层 Windows 函数，它还拥有创建 Windows 原生 GUI 的函数。机制是相同的，只是调用的函数不同。

这两个工具包尤其是对于 Win32 API 来说，其共同弱点就是复杂性。而且它们被严格限于自身的操作系统。如果知道永远不会支持任何其他操作系统的话，这可能就不是问题了。Cocoa 选项至少提供了一个有用的开发工具集，而 Windows 选项还不如它丰富。

对于原生 Windows 开发，一个更好的选择是 IronPython。它是 Microsoft .NET 实现的 Python 版本，支持标准 .NET 语言的 Microsoft Visual Studio 开发环境。但是它仍然只限于 Windows (Mono 是 .NET 为其他平台的克隆，但是它并没有广泛地用在桌面应用中)。

MacPython 网址是 <http://homepages.cwi.nl/~jack/macpython/>。IronPython 网址是 <http://ironpython.net/>。

4.11.5 Dabo

Dabo 与其他描述的工具具有显著的区别，因为它不仅是一个 GUI 工具包。Dabo 是一个全功能的应用框架和工具集。它专门用于数据库应用，比如在商务中常见的数据库。它有一组 GUI 小控件，当前是基于 wxWidgets 基础但是为 Dabo 做了修改(理论上说，也可以使用其他 GUI 工具包，但是开发团队已经找到了更加有效率的方式来使用它！)。在 GUI 的顶端，它提供了一组类，这些类包含事务逻辑并缩小用户界面与数据层之间的距离。数据层可以使用任意流行的数据库，包括 SQLite。

也可以在 Dabo 应用中创建复杂的逻辑，但是它的本质功能是创建窗体。这个工具提供了视图和编辑基础数据的功能。0.9.12 版本在 2013 年 6 月发布。Dabo 网址为 <http://www.dabodev.com/>。

你现在已经了解了一些基础知识，尤其是关于用户界面选项的知识。在使用 Python 创建真实世界桌面应用时，你可能会碰到一些其他话题：存储配置数据和本地化。现在将介绍这些话题。

4.12 存储本地数据

在第 3 章，你已经看到了各种用于存储数据的策略。在本章的开头介绍了如何将应用组织为层的结构。层结构的栈底是一个数据层。数据层用于管理应用的核心实体。核心实体并不是你唯一需要存储的数据。通常你也需要存储应用本身相关的数据。这个配置数据是特定于个别用户的，或者特定于本地计算机系统的。因此被称为本地数据。通常，它被存储在配置文件中或作为环境变量。你在第 2 章已经了解了如何读取此类数据。在本节中，你将接触到应用可能需要维护的各种本地数据，以及可用的存储方法。

应用有多种不同类型的配置数据，其中一些用于让应用首先工作起来，比如服务器的网络地址或数据文件的位置。这些数据通常特定于给定的安装版本或计算机系统，而非特定于个别用户。

其他类型的配置数据包括用户偏好的东西。例如，用户可能会控制用户界面布局、使用的颜色、屏幕上 Windows 的位置和大小等。另一种常见的用户配置数据是所选的帮助应用，比如用户喜爱的文本或图片编辑软件。其中一些细节可以放在一个偏好对话框中，用户可以编辑和存储这些内容。其实，它甚至可以为同一应用的不同使用场景存储不同的偏好。

应用本身可能存储其他设置，这样它可以在重新启动时恢复到上一次的状态。这些设置可能包括最后打开的文件、当前打开的窗口和对话框，以及每个窗口的屏幕坐标。

最后，你可能想要保存的数据信息是关于应用如何起作用的。尤其是，可以记录错误情况或意料之外的输入。这通常被称为日志并涉及将信息保存到日志文件中。之后可以检查日志文件或者作为调试进程的一部分，或者用于提高设计的效率。

4.12.1 存储特定于应用的数据

可以将应用特定的数据存储在本机计算机中，或者存储到应用的所有实例都可以访问到的本地网络位置中。这产生了计算机如何知道查找位置的问题。通常，这个问题的解决方案是设置一个环境变量或使用存储在应用的启动文件夹中的本地配置文件。然后可以设置它为安装进程的一部分，甚至可以为它提供一个启动参数。

在网络上存储这类信息的优势在于它是被分享的。这样所做的任何改变，比如数据库被移动了，都可以立刻被网络上所有安装的实例检测到而不需要手动配置每个机器或用户配置。它同样顾及到在系统失败事件发生时可以使用的备份配置。而通过改变一个环境变量或在本地计算机的配置设定，可以实现在最小的故障时间后访问到新的中央配置。

物理存储媒介相对不重要，因为数据相对固定并且通常只在应用启动时读一次。使用纯文本、XML，甚至 Windows INI 格式的简单配置文件可能就足够了。Python 为创建和读取所有这些文件提供了工具，可以从第 2 章查看详情。

4.12.2 存储用户选择偏好

用户偏好几乎永远被存储在本机计算机中，并且通常被存储在用户的主目录中的一个配置文件中。有时，应用会把它存储在主数据库中，尤其是当数据库已经包含了大量用户数据时。使用数据库的弱点在于应用只有在数据库可访问时才可以访问偏好。但如果用户是移动的，这可能会有问题。令人不安的是，用户会发现应用外观或功能会根据他们是否联网而不同。对于这种类型的数据，本地存储绝对是更好的选择。

如果使用了标准的文件名并且地址是主目录，定位数据就很直接。因为主目录几乎是永远可以获得的，不管是作为环境设置还是作为用户数据库值(可以从第 2 章查看如何获得用户细节，比如主目录)。

格式通常是一个使用 Windows INI 或者 XML 格式的文本文件。如果设置的数量非常大，那么一个使用 SQLite 的小型本地数据库可能会适合，但是这可能会与主应用数据存储分隔开。

在处理用户偏好时，另一个需要考虑的因素是如何设置和修改这些偏好。如果设置很少而且本质上很简单，那么生成一个默认偏好文件并让用户手动修改文件是可行的。但是如果用户不熟悉文本编辑器，并使用富格式文字处理器来编辑这个文件，这就会产生风险。这可能导致应用不能读取配置文件。然而，如果用户很熟悉文本编辑，比如开发者或系统管理员，那么这个方法很奏效。如果配置数据不简单，或者用 XML 或类似格式存储在结构化的文件中，那么用户编辑就不是很合适而且容易出错。这时需要在应用本身加入一个偏好对话框。

存储并命名配置数据

一些操作系统或环境有首选的存储配置数据的方式。Microsoft 推荐 Windows 应用使用 Windows 注册表(为此，Python 提供了 winreg 模块)。注册表可以为单独用户或计算机存储

数据。注册表的弱点在于它是特定于 Windows 的。当代码试图运行在多个操作系统时，需要有两种存储机制。所以很多开发者喜欢标准化配置文件，即使对于 Windows 也是如此。

X Windows 系统有它自己的配置数据库(通过 `xrdb` 程序管理)，而它的可配置数据是一种特殊的格式。所有 X 应用的本地设置都被存储在用户主目录的 `.Xresources` 文件中。幸运的是，如果正在 Python 中创建基于 X 的程序，可能会使用之前讨论的 GUI 工具包，而这些工具向你隐藏了这些细节。然而，如果应用使用基于 X 的程序作为帮助应用，则可能需要读取或修改这些 X 配置文件。这应该是一个脚本化的挑战，而第 2 章中介绍的原理会很适用。

最后，大多数操作系统为配置文件推荐了命名习惯。在 UNIX 系统上，它们通常以英文句号开始(这样使它们在普通文件列出工具中不可见)，以字符 `rc` 结束，并且通常被存储在用户主目录(为用户设置)和应用目录(为默认和系统设置)中。例如，`vim` 文本编辑器使用 `.vimrc` 来存储用户偏好。越来越多的应用把配置文件存储在隐藏目录下。而这也是 `freedesktop.org` 标准推荐的地点。在 Windows 上，配置文件的格式通常是 Windows INI 格式，扩展名为 `.ini`。

4.12.3 存储应用状态

存储应用状态在本地数据存储中是最不标准化的。存储的位置和格式由开发者决定。一些应用对于是否为用户偏好保存状态做了一些选择，而有些应用则自动完成选择，但是大部分并不保存状态(除了可能保存最近访问的文件列表)。需要判断你想要存储多少状态信息、存储在哪里，以及使用什么格式。

为了保持应用行为的一致性，应该选择本地存储选项，这样应用会像预期的那样工作，即使没有连接到网络上。然而，需要小心对错误的处理，因为如果应用被在线关闭但是后来被脱机打开，许多之前使用的资源可能就无法访问了。需要有一个回退工作的配置，当发生错误时，可以使用这个配置。

状态数据的格式通常很复杂，因为它可能涉及多窗口，以及窗口内的标签和控制设置。这几乎不可避免地需要一个丰富的存储格式，比如 XML。另一方面，如果只保存打开文件历史，使用一个简单的文本文件可能就足够了。甚至可以将它附加到用户在配置文件中的偏好数据上。

4.12.4 记录错误信息

你通常想要记录意想不到的事件或输入，这样可以在之后分析它们。这可能发生在测试期间，在一个系统失败之后，或者是作为一个持续的改善应用的行为。常用的方式是把消息记录在日志文件中。日志文件可能是一个单独的文件，它会持续地增长。或者更常见的是一个基于日期的文件。旧文件的日常管理(归档或删除旧文件)可能是通过手动、自动或一个 `shell` 脚本来完成。

为了辅助这个过程，Python 提供了 `logging` 包。它可以在一个标准文件中生成带有不

同日志级别的标准信息(也就是它可以用不同的分类标记一个信息: 调试、信息、警告、错误和紧急)。这个包非常灵活, 并且允许使用许多不同的配置选项来控制它如何工作。它的基本用法很简单, 可以在需要时扩展它们的功能。

在最基本的级别, 你导入模块, 然后调用适用于提到的级别的几个日志方法中的一个。如下:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> logging.info('Heres some info')
INFO:root:Heres some info
>>> logging.error('Oops, thats bad')
ERROR:root:Oops, thats bad
>>> logging.critical('AAARGH, Its all gone wrong!')
CRITICAL:root:AAARGH, Its all gone wrong!
```

在使用任何日志方法前调用 `basicConfig()` 是很重要的。级别设置表明了信息显示的最低级别。DEBUG 是最低级别, 所以所有内容都被打印了出来。除了设置级别, 还可以指定一个输出文件名。

也可以指定日志消息的格式, 包括像消息日期、产生消息的文件和函数等这些消息(选项的相关文档都在日志文档的 `LogRecord` 部分)。由于默认格式不包含任何日期或时间信息, 因此常常希望进行一些设置, 如下所示:

```
>>> import logging
>>> logging.basicConfig(format="%(asctime)s %(levelname)s : %(message)s")
>>> logging.error('Its going wrong')
2014-04-24 16:12:44,832 ERROR : Its going wrong
>>> logging.error('Its going wrong')
2014-04-24 16:12:54,415 ERROR : Its going wrong
>>>
>>> logging.critical('Told you...')
2014-04-24 16:13:08,431 CRITICAL : Told you...
```

也可以在 `basicConfig()` 中使用 `datefmt` 参数, 这样就可以使用与 `time.strftime()` 相同的选项来改变日期格式。下面是一个小示例:

```
>>> import logging
>>> logging.basicConfig(format="%(levelname)s: %(asctime)s %(message)s",
                        datefmt="%Y/%m/%d - %I:%M")
>>> logging.error("It's happened again")
ERROR:2014/04/24-04:21 It's happened again
>>>
```

还有很多其他可以做的事情, 但在此描述的基础用法对于除了最大项目之外的所有项目都应该足够了。

4.13 理解本地化

本地化是指为了让计算机应用可以用在不同地点所需要的各种行为。这包括一些特性，比如时区差异、日期和时间格式化、货币标志、数字格式化，以及必要的语言差异。在极端情况下，它可能需要一个新 UI 布局来处理阅读方向，比如从右向左。你的电脑操作系统应该有很多这些特性。它们都通过一个配置设置来控制，通常是在操作系统第一次安装时被创建。大部分用户从来没有改变这个设置，也因此认为它就是这样的，从来没有仔细研究过它。作为一个程序员，你的代码可能需要运行在不同的电脑上，每个可能都需要本地化选项。

在本节中，你将学到 Python 如何使用 `locale` 和 `Unicode` 字符集支持本地化。`locale` 只是一个代码值，它代表标准化的本地设置组(时区、日期格式、货币等)。`Unicode` 是一个用来表示不同字母表字符(比如拉丁文、阿拉伯文、中文等)，以及不同的符号(比如标点符号和数学符号)的国际化标准。

`Unicode` 非常有用，因为它让我们可以使用不同的字母表。但是如何将应用中的字符串翻译成不同的语言呢？这个过程被称为国际化。为此，有一个标准行业过程使用了名为 `gettext` 的机制。这个机制产生了特定于语言的文件，该文件包含了从源字符串到不同语言版本的映射。Python 通过 `gettext` 模块支持这个机制。本地化包含使用 `gettext` 选择正确字符串翻译的能力。

4.13.1 使用区域设置

Python 通过 `locale` 模块支持不同的区域。模块工作的方式很复杂，并且使用了分层的方法。但是在大多数情况下，并不需要知道这些。可以使用非常小的一部分子集，然后为你的用户选择正确的区域。

当程序启动时，它通常会被设置为默认的 C 区域(尽管可能有时不是这样的，而且本地配置设置可能已经修改了它)。然而，通常想要把区域设置为用户已经选择的区域。做这件事情的方法是用空的区域参数调用 `locale.setlocale()` 函数。这会选中系统区域。大多数情况下，这就是需要做的。强烈建议你只在程序中做一次该操作，并且在代码开始的地方做。

如果想知道设置的区域，可以使用 `locale.getlocale()` 获得本地详情(将在下一节中介绍该操作，你将明白如何将程序的字符串翻译成本地语言)。

遗憾的是，设置区域还没结束。如果想要这个改变生效，需要对你的代码做一些改变。特别的，在标准库和内置函数中，有一些类型转换和比较操作并不在意区域。为了绕过它，`locale` 模块提供了一些可选方案。在其他情况中，如果给他们正确提示的话，标准函数会理解区域。例如，如果使用合适的格式化指定比如为本地日期使用 `%x` 并为本地事件使用 `%X`，那么 `time.strftime()` 函数可以把时间格式化为本地风格的字符串。同样的，如果想要使用本地特定的格式打印数字的话，则需要在字符串 `format()` 方法中指定 `n` 风格而不是 `d`、

f 或 g。

下面的解释器会话演示了这些特性：

```
>>> import locale as loc
>>> import time
>>> loc.setlocale(loc.LC_ALL, '')
'English_United States.1252'
>>> loc.currency(350)
'$350.00'
>>> time.strftime("%x %X", time.localtime())
'4/22/2014 7:44:57 PM'
>>>
```

在 cygwin 会话中重复这些操作并设置为 en_GB 区域，可以看到一些差别：

```
>>> import locale as loc
>>> import time
>>> loc.setlocale(loc.LC_ALL, '')
'en_GB.UTF-8'
>>> loc.currency(350)
'£350.00'
>>> time.strftime("%x %X", time.localtime())
'22/04/2014 19:32:23'
```

系统非常清楚地指定了不同的区域。货币使用了合适的符号，而且日期和时间与 UK 版本有很大的区别。它使用了 24 小时时钟格式并且日期中的天和月发生了调换。

现在可以试试一些转换和比较函数了。这些函数在 UK 和 U.S.英语中的工作方式是相同的，所以现在不必做一些比较。

```
>>> print("{:n}".format(3.14159))
3.14159
>>> print("{:n}".format(42))
42
>>> loc.strcoll("Spanish", "Inquisition")
1
>>> loc.strcoll("Inquisition", "Spanish")
-1
>>> loc.strcoll("Spanish", "Spanish")
0
```

这些示例显示了字符串格式化指定器是如何同时为整数和浮点数工作的。locale.strcoll() 字符串比较示例是非常有用的，因为它们在字符排序中采用了特定于区域的想法。如果第一个字符串有更“高”的值，返回值就是 1，如果字符串的值更“低”，返回值为-1，而如果两个参数相同，则返回 0。

local 提供了转换函数，这些函数在特定情况下是很有用的：atoi()、atof()、str()、format() 和 format_string()。

4.13.2 在 Python 中使用 Unicode

计算机将数据存储为二进制数。字符被映射到这些数字上，这样当计算机打印一个字符型字符串时，计算机会将内存中的数字数据映射到屏幕上的字符字符串。回到计算机的初期，字符用最少 5 位而通常是 7 或 8 位来表示。所有这些存储都可以适用于 8 位字节的存储方案。所以它的兼容性非常好。遗憾的是，单个位只能最多产生 256 个不同组合。这对于在计算机起源的西方国家使用的拉丁字符很好用。但是如果把世界作为一个整体时，这就远远不够了。一直以来，每个国家和企业都在发明子集的编码系统，而当软件要在不同区域使用时，软件工程师已经编写了大量代码来适应这些。解决方案就是 Unicode 标准。

Unicode 是一个 32 位字符目录，它可以存储大量字符。Unicode 字符是通过码位(code points)来表示的。码位就是映射到字符的数字值。码位的描述使用格式 U+xxxx。U+表示它是一个 Unicode 码位，而 xxxx 是十六进制数，用来表示码位在 Unicode 中的位置(码位最多可以是 8 位十六进制数而不仅仅是 4 位)。之后码位被映射到拥有描述性名字的字符上。因此，字符“A”被列为“U+0041 LATIN CAPITAL LETTER A”。Unicode 编码的 Python 表示为 4 位的\uxxxx 或 8 位的\Uxxxxxxxx。由于仅仅是把 U+替换为\u或\U，这种形式显然是 Unicode 格式的一种简单转变。名字和码都可以在 Unicode 网站上找到：<http://www.unicode.org/Public/UNIDATA/NamesList.txt>。要认识到 Unicode 只是定义了字符，而不是它的外观，这一点很重要。你在电脑屏幕上看到的或打印到纸上的字符被称为标志符号。它是那个字符以某种字体的图形化表示。Unicode 没有指定字体、大小或任何其他外观细节，它只指定了实际的字符。

Unicode 数据必须作为一组字节存储在计算机上。你可能还记得，它只能存储从 0-255 的值。Unicode 的最简单转换或编码被称为 UTF-32，它是一个从 Unicode 码位值到一个 32 位数的——映射。这非常好理解，但是需要为每个字符使用 4 个字节。这会消耗非常大的内存和带宽。为了节省控件，还有其他两个编码。UTF-16 使用 16 位块来表示大多数字符，但是有一个选项可以在表示一些很少用到的字符时把它扩展为两个 16 位块。Microsoft Windows 默认使用 UTF-16。



注意：在 UTF-16 中，扩展名被称为替代域，通过一个包含范围在 0xD800-0xDFFF 的值的块来指定。UTF-8 使用了不同的方案。如果一个字节拥有大于 128 的值，这表示它是多位序列的一部分。

UTF-8 将最常用的字符存储在单个 8 位块中，但是可以在表示不常用字符时被扩展为使用 2、3 或 4 个块。如果在使用恰好的字符集尤其是拉丁字母表时，这让 UTF-8 成为最兼容的格式。UTF-8 还有一个很方便的特性。它将原本的 ASCII 字符集作为它字节最低部分的字符集。这让与旧的非 Unicode 应用的联合工作变得轻松很多。UTF-8 是 Python3 的

默认编码方式。

这很复杂，那么 Python 可以如何帮助你处理这些呢？尽管你在需要时可以修改，但是 Python3 使用 Unicode 字符串并将 UTF-8 作为它的默认编码。修改编码的方式是将一个特殊的注释放入代码的首行，如下：

```
# -*- coding: <encoding name> -*-
```

在这里，encoding name 就是你选择使用的编码，通常是 utf-16 或 ascii。

也可以在字面量字符串中使用 Unicode 字符。甚至可以使用它们的长名字，例如：

```
>>> print('A')
A
>>> print('\u0041')
A
>>> print("\N{LATIN CAPITAL LETTER A}")
A
```

它们都打印同样的字符 A。

可以使用字符串的 encode() 方法来获得用于存储数据的原生字节：

```
>>> print("\u00A1Help!")
!Help!
>>> print("\u00A1Help!".encode('utf-8'))
b'\xc2\xa1Help!'
>>> b'\xc2\xa1Help!'.decode('utf-8')
'!Help!'
>>>
```

注意，第二个版本打印了表示为 \u00A1 的倒感叹号的 UTF-8 双字节表示形式。第三行使用了 encode() 的配套方法 decode() 把编码字节转换回字符串。

迄今为止，你已经了解如何表示单个 Unicode 字符、如何改变 Python 使用的默认编码方式，以及如何使用 encode() 把字符串转换成它的字节表示形式和使用 decode() 把字节转换为 Unicode 字符串。在大多数情况下，这就是需要了解关于 Unicode 和 Python 的内容。解释器为你做了大部分的工作。也可以接着定义自己的编码方式。但是那需要一些更详细的学习和 codecs 模块的学习。在大多数情形中，你真的不需要做这些。

在本节中结束时，你应该意识到了 unicodedata 模块。它在交互式提示符中是非常有用的。它可以作为一种找到关于 Unicode 字符的方式。结合在 Unicode 网站发布的数据，你应该能够回答在日常写代码时碰到的大多数问题。

通过使用 unicodedata 模块，可以得到一个给定字符的 Unicode 名字和类别。然后根据这些可以在网站查到更多细节。假设你刚好存储了一些你认为包含 Unicode 字符串的数据，而且想要知道它包含的字符，可以试试这个模块：

```
>>> data = b'\xd9\x85\xd8\xb1\xd8\xad\xd8\xa8\xd8\xa7 \xd8\xa3\xd9\x84\xd8\xa7\x
\xd9\x86'
>>> print(data.decode('utf-8'))
```


ألان مرحبا

```
>>> for ch in data.decode('utf-8'):
...     print(ord(ch), ud.name(ch))
...
1605 ARABIC LETTER MEEM
1585 ARABIC LETTER REH
1581 ARABIC LETTER HAH
1576 ARABIC LETTER BEH
1575 ARABIC LETTER ALEF
32 SPACE
1571 ARABIC LETTER ALEF WITH HAMZA ABOVE
1604 ARABIC LETTER LAM
1575 ARABIC LETTER ALEF
1606 ARABIC LETTER NOON
>>>
```

在这里有一个字节字符串。你怀疑它是 UTF-8 字符，并且想要知道它是什么类型的数据。可以试着对它进行解码，来检查它是否是有效的 UTF-8，而这成功了。但是你不认识打印出来的字符集。然后导入了 `unicodedata` 模块并在数据上执行了一个 `for` 循环来打印每个字符的长 Unicode 名。这样，很明显，数据是阿拉伯语。

4.13.3 使用 gettext

为了使用 `gettext` 机制翻译程序中的字符串，需要执行几个标准的步骤。首先，必须使用 `gettext` 函数来确定代码中你想要翻译的字符串。其次，执行一个工具将那些字符串抽取到模板文件 `messages.po` 中。此后，需要基于 `messages.po` 生成翻译文件，理想情况下，是通过雇佣一些翻译者或者可能通过信任 Google 翻译或类似的工具。再次，使用另一个工具将翻译文件转换为 `gettext` 使用的特定于语言的 `.mo` 格式，比如英文版本的 `messages_en.mo`。最后，需要将文件夹连同其中的 `.mo` 文件与你的翻译文件放在一起。根据操作系统的不同，有各种不同的可用工具。对于 Windows 用户来说，在 Python 发布包的 `Tools/i18n` 文件夹下有几个脚本。对于类 UNIX 系统，也有 Python 可用的操作系统工具。

现在查看一个简单的示例——普遍使用的“Hello world”脚本。第一步是使用 `gettext` 模块和它的函数标记出程序中需要翻译的字符串。首先，创建一个新的 Python 代码文件 `gettext_demo.py` (或者从 `zip` 文件的 `gettext` 文件夹中加载它)：

```
import gettext
import locale as loc

# Set up the locale and translation mechanism
#####
loc.setlocale(loc.LC_ALL, '')
filename = "res/messages_{}.mo".format(loc.getlocale()[0][0:2])

trans = gettext.GNUTranslations(open(filename, 'rb'))
trans.install()
```

```
# Now the main program with gettext markers
#####
print(_("Hello World"))
```

这个脚本像之前部分讨论的那样设置了区域，并使用它动态地创建了将要使用的翻译文件的名字。只创建了一个英文翻译文件，所以可以硬编码这个名字，但是使用 `getlocale()` 演示了当有多个可用语言时应该如何做。

接下来，实例化了一个 `gettext.GNUTranslations` 对象并对此对象执行 `install()` 操作。这激活了函数 `_()`，之后使用该函数包围所有需要翻译的字符串。在这个示例中，它就是字符串 “Hello World”。

下一步是生成 `messages.po` 模板文件。如果是在一个类 UNIX 的操作系统上，可以按如下方式运行工具 `xgettext`：

```
$ xgettext gettext_demo.py
```



注意：Windows 上与 UNIX 工具相对应的是 `pygettext.py` 和 `msgfmt.py`。这两个文件都在标准 Python 安装包的 `Tools/i18n` 文件夹中。

现在应该在 Python 文件的同一文件夹中找到一个 `messages.po` 文件。在你的文本编辑器中打开它，然后将字符串 `CHARSET` 替换为 `UTF-8` 并在底部的空字符串中插入 “Hello World” 的翻译。为了证明这条翻译已经生效，应该使用类似 “Hello Beautiful World” 的东西。但通常对于英语来说，你只需要重复同样的字符串(你还可以填充其他一些 `metadata` 字段，但是本示例中并不需要它们，所以可以忽略它们)。现在将文件保存为 `messages_en.po`。代码如下：

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""

"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2014-04-22 18:05+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8 \n"
```

```
"Content-Transfer-Encoding: 8bit\n"
```

```
#: gettext-demo.py:14
msgid "Hello World"
msgstr "Hello beautiful world"
```

接下来，创建 `res`(保存资源的)文件夹。已经在 Python 脚本中指定了这个文件名称。虽然 `res` 并不是一个严格的名称，但它是一个很有道理的命名约定。现在，执行另一个名为 `msgfmt` 的工具，如下：

```
$ msgfmt -o res/messages_en.mo messages_en.po
```

注意不同的文件结尾！上面的代码创建了翻译文件，告诉 `gettext` 在执行脚本时读取这个文件。现在已经准备好运行该示例了：

```
$ python3 gettext_demo.py
```

可以看到，打印出来是被翻译的字符串“Hello beautiful world”而不是代码中的初始“Hello World”。这证明翻译生效了。通过为每个新语言重复执行 `xgettext` 之后的步骤，可以继续生成更多翻译文件。为了测试它们，需要改变本地语言设置。

本地化是一个复杂但持续增长的研究领域。然而，如果计划将你的应用发布到多语言用户或发布到不同的国家，你一定不能忽视它。

4.14 本章小结

在本章中，你已经看到了在层中组织应用将数据处理与核心或业务逻辑以及展示分开的强大之处。特别是，你看到了如何在相同的核心逻辑层和数据层上创建多种用户界面。在这个过程中，探索了数个命令行界面的变形，包括用户交互的不同风格以及强大的命令行选项。

你也看到了如何使用 Python 中的标准 GUI 工具包 Tkinter 创建 GUI 应用，以及可以提供更多小部件和改善外观的辅助模块。在本节结束时，基于已有的数据层创建了一个用户界面，使用了很多之前探索过的特性，但使用的是面向对象而不是过程的风格。最后，回顾了一些可选的第三方 GUI 框架。相比 Tkinter，这些框架提供了更加强大的功能。可能希望为 GUI 应用获得更多这样的功能。

本章讨论了创建供他人使用的应用时存在的一些问题。这涉及可以存储的各种类型的非核心数据(比如配置值)，以及每种类型可用的选项。你也了解了如何使用 Python 的 `logging` 模块来记录重要事件，如何管理 `logging` 的级别，以及如何存储它。

在本章的最后，介绍了围绕着用户本地化应用的一些问题。这包括为货币使用本地化的设置、时间格式以及不同的字母表。Python 支持 Unicode 字符集，并且可以使用 `encode()` 和 `decode()` 方法将字符串转换为它们的原始字节形式或从字节转换回字符串。最后介绍了在应用中显示不同语言的 `gettext` 机制。

练习

- 1. 创建 Game 类，转换 oxo-logic.py 模块以反映 OOP 设计。
- 2. 探索 Tkinter.filedialog 模块，从用户获取一个文本文件名，然后将文件显示在屏幕上。
- 3. 将第一个 GUI 示例中的标签替换为一个 Tix 的 ScrolledText 小部件，这样它会显示 Entry 小部件中所有实体的历史信息。
- 4. 重写第一个 GUI 示例，让它兼容 gettext，并使用控件上的不同文本生成一个新的英语版本。

本章所学知识

主 题	关 键 概 念
桌面应用	主要在用户本地计算机上运行的程序。通常是交互式的程序，比如修改数据或玩游戏
数据层	应用存储、修改和获取数据的部分。它不清楚任何应用逻辑或业务规则，也不知道数据如何被展示
核心逻辑	应用处理数据的部分，其中包含复杂的算法和业务规则。数据会通过数据层获取或写入。逻辑并不关心结果如何被展示，只是在意产生正确的数据
表示层(又称用户界面)	应用与用户交互的部分。这部分的任务是以一种清晰的方式展示数据，并通过有逻辑的、明显的方式让用户使用应用的功能。表示层不应该依赖底层函数提供的数据的准确性，它只关心如何展示它得到的数据(一些基本的数据校验是合理的，比如确保给定字段包含一定范围内的整数)
本地化	是一种进程和机制，允许多个用户不管在什么地方都可以使用同一个应用，并且看到他们能够识别的输出。也就是说，应用需要兼容结构化数据的本地布局，比如日期和货币
Unicode	一组国际公认的、用来表示字符集的标准集。Unicode 不在意它表示字符的形状(它们的形象)，只在意单个字符的内容或意思
国际化	将字符串翻译为可显示的形式，这样不同语言的用户都可以明白

第 5 章

Python 在 Web 中的应用

本章主要内容:

- 理解 Python 的 Web 工作原理
- 使用 Python 创建 Web 应用
- 将 Web 应用连接到数据库
- 创建 API
- 解析和操纵 Web 应用中的数据

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/pythonprojects 的 Download Code 选项卡下找到。第 5 章代码在 Chapter 5 download, 名为 Chapter5.zip, 每个文件都是根据本章提到的代码文件名命名的。

迄今为止, 已经在本地计算机上使用 Python 研究和解析数据。但是, 当想要使用 Python 进行远程或跨 Web 操纵数据时, 需要如何做呢?

Python 是一门非常强大的语言。除了本地的系统管理任务或文件系统任务, Python 还可以使用 REST 和 XMLRPC 跨网络处理一些任务。本章将会涉及这些内容, 但重点是远程使用 Python 的最常用的方法: 使用 REST 的 HTTP。

在本章中, 会用到一些 Python 中更流行的 Web 技术, 包括 HTTP 和 REST。会使用它们创建一个 Web 应用并连接到数据库。将使用技术(比如 Flask 和 SQLite)来完成 Web 应用。最后会创建一个应用编程接口(API), 并学习在 Web 应用中如何解析和操纵数据。

REST 代表 Representational State Transfer。它是某个 Web 服务技术和架构风格的有趣的名称。这个服务不依赖组成部分或协议语意的执行。相反, 它关心如何与数据交互(在其约束中发送和接收数据)。我们会在使用 Flask 创建 API 的部分解释 RESTful 架构。现在, 只需要知道 REST 是 Web 服务架构的构成部分。



注意：因为前端 Web 开发的基础不在本书的范围之内，我们已经为你提供了所需要的文件，这样可以拥有一个良好的界面。我们会聚焦在 Python 的中间层和如何将前端连接到数据库和 Web 服务器上。我们只是浅显地接触所提供的 JavaScript。如果要专注于 Python Web 开发，那么推荐你熟悉现代前端技术。

5.1 Python 在 Web 中的应用

你可能也知道，Web 是由多种技术组成的，而不只是一种。图 5-1 展示了一个现代 Web 应用的高层次的概览。

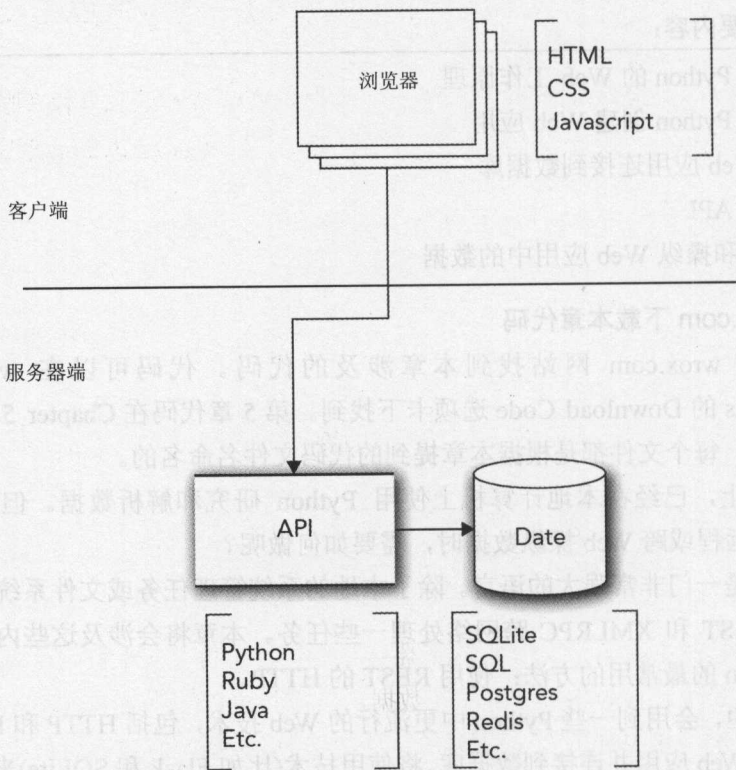


图 5-1 注意 Web 应用的两端：服务器端和客户端。服务器端数据是通过 Web 服务器提供的，比如 Apache 或 Nginx

如你所见，Web 应用的前端包含一个处理 HTML、CSS 和 JavaScript 的浏览器。在此，中间层是 Python，但大多数情况下，它在某种程度上是脚本语言，如 Ruby、Perl、PHP 或

JavaScript。后端会处理数据库(SQLite)和 Web 服务器，或 HTTP Daemon(通常，Nginx 或 Apache)。尽管后端和前端基础不在本书的讨论范围内，但是你将学习如何设置一个简单的服务器。



注意：如果想要把这个服务器公用，强烈推荐你了解任何不熟悉的技术基础。所以，请在此之前阅读相关文档来了解这些内容，并学习如何设置一个面向公众的服务器。

接下来，我们解释这些部分是如何协同工作的。

5.1.1 Web 应用的组成部分

有多种方法可以跨网络来服务数据。Python 几乎可以处理所有这些方法。我们会简略地了解 Python 在服务器端的常见用途。然后会创建一个 Web 应用。这是在 Web 中使用 Python 的最流行和最常用的方式。

任何网络请求的标准结构如图 5-2 所示。“服务器接收请求”和“服务器向请求者返回数据”之间会发生什么事情呢？这种数据之间的传递是通过 Python 来完成的。

发送请求



服务器接收请求



服务器向请求者返回数据

图 5-2 网络请求的结构

由于浏览器不能自带所有的解释器——显然，我们不想要它们自带编译器，我们需要寻找一种方式来连接静态页面和动态数据，这样就可以拥有交互式动态网页。我们有一个 Web 服务器来提供 Web 应用所需要的文件，但是服务器如何知道所使用的是什么文件呢？服务器如何分清应用是用 Python 实现的而不是用 Ruby 实现的呢？

Web 服务器(比如 Apache)通过 WSGI(Web Server Gateway Interface)知道这些情况，并且 WSGI 可以运行 Python 文件。本书不会探索错综复杂的 WSGI。你只需要知道最新的 Web 服务器支持它即可。如果想要在一个服务器上运行 Python，该服务器就需要有可用的 WSGI。如果想要更加深入地理解服务器操作以及 DevOps，建议你对更底层的技术做一些

研究,比如 WSGI。为此,我们会使用一个框架,该框架已经为我们处理好了服务器架构并且可以使用 WSGI。在讨论 Python 在 Web 上的应用时,你会看到这个术语在一直被引用。所以希望你记得它并理解它在很大型项目中所起的作用。

5.1.2 客户端-服务器关系

什么是客户端?在 Web 开发中,客户端通常是指一个 Web 浏览器。它向你的 Web 服务器请求文件。然而,你会看到客户端也指向第二个服务器请求数据的另一个服务器。在本章,客户端基本上是一个浏览器,换言之,也就是 Web 应用中用于显示返回数据的部分。这也是 Web 应用中用户可以交互的部分。如果曾经听说过术语“客户端 JavaScript”但从没真正理解它,这应该可以解除你的困扰:客户端 JavaScript 依赖客户端并且直接在客户端(一个 Web 浏览器)中执行操作,而不是在服务器上。例如,当特定事件(比如单击按钮来改变背景颜色)被触发时改变网页的颜色或风格。如果需要的话(没有 Internet 连接),这些功能实际上从来不与服务器交互,只是在本地使用。

那么,什么是服务器端 JavaScript 呢?可能你已经猜到了,它是在服务器上执行的 JavaScript,通常通过客户端事件触发,比如 GET 请求或客户端 JavaScript。在本书中,我们不会涉及任何服务器端 JavaScript,因为正在使用的是 Python。Python 是我们选择的服务器端语言,这也是我们要学习的内容。这是应用中从客户端获取行为并展示魔法的部分。

5.1.3 中间件和 MVC

中间件是 Web 开发中一个很新的术语。它是指技术栈部分。它负责从前端(客户端)获取数据,处理它,传入和传出数据库(或其他运行的服务),然后把它发回前端。基本上,系统或应用的逻辑应该在中间层,数据应该在数据层,而风格应该在前端。在前端做很多数据逻辑是很不理想的,最好放在中间层做。

所有这些部分实际上有一个非常好的名称——模型-视图-控制器 (MVC)。大多数现代框架都在某种程度上使用 MVC 架构。MVC 的优势是客户端不需要处理应用的逻辑,而逻辑不需要处理数据模型。你甚至可以在设置数据模型后忘掉它!

如果在一个大项目中工作,并且有网页设计师编写 HTML、CSS 和一些 JavaScript,这些让你的应用看起来非常成熟美观。而你还应该在数据层拥有非常聪明的数学家,因为需要对从成熟美观的客户端获得的数据进行精确的数字计算。如果把逻辑代码放在前端文件中,会发生什么呢?假设将计算放在网页设计师工作的模板中,而在他们想要操作一些东西时,他们认为你的计算有问题。这很不理想,不是吗?MVC 在某种程度上通过把数据层从控制器分离出来解决了这个问题。可以编写控制器来做精确计算,然后使用网页设计师所编写的视图来实现成熟美观的 Web 应用。

那么,这些在 Python 世界意味着什么呢?答案是 MVC。你应该简要了解一下这个架构。在本章,我们会使用一个利用了 MVC 架构的框架。所以现在你应该更能理解我们所做的很多决定。

5.1.4 HTTP 方法和头信息

HTTP 代表超文本传输协议(HyperText Transfer Protocol)。通常,这是用来在 Web 上通过 Web 浏览器传输数据的协议。当一个客户端发起请求时,它就会使用这个 HTTP 协议。那些使用过超文本标记语言(HTML)进行 Web 编程的人可能会很熟悉它。HTTP 被用来传输 HTML。

HTTP 方法是 Web 上的动词。两个最基本的方法是 GET 和 POST。它们可以实现你所期待的事情。当向 Web 服务器发送一个 GET 请求时,是在从服务器请求数据。在每次向浏览器加载网页时,都会使用这个请求,或 GET。所以,当访问 twitter.com 时,浏览器会向 Twitter 服务器发送 GET 请求并请求信息。

当编写一个 tweet 并按下 Tweet 按钮时,又发生了什么呢?在向服务器发送数据时,调用了 POST 方法。这告诉服务器正在试图向服务器 post 数据(在创建应用的过程中,我们会少许接触一些其他方法调用)。



注意: 如果打算使用 Python 进行 Web 开发,建议你应该熟悉所有可用的 HTTP 方法。

那么 Web 服务器是如何知道调用的是哪个方法呢?它是通过 HTTP 头知道的(如图 5-3 所示)。

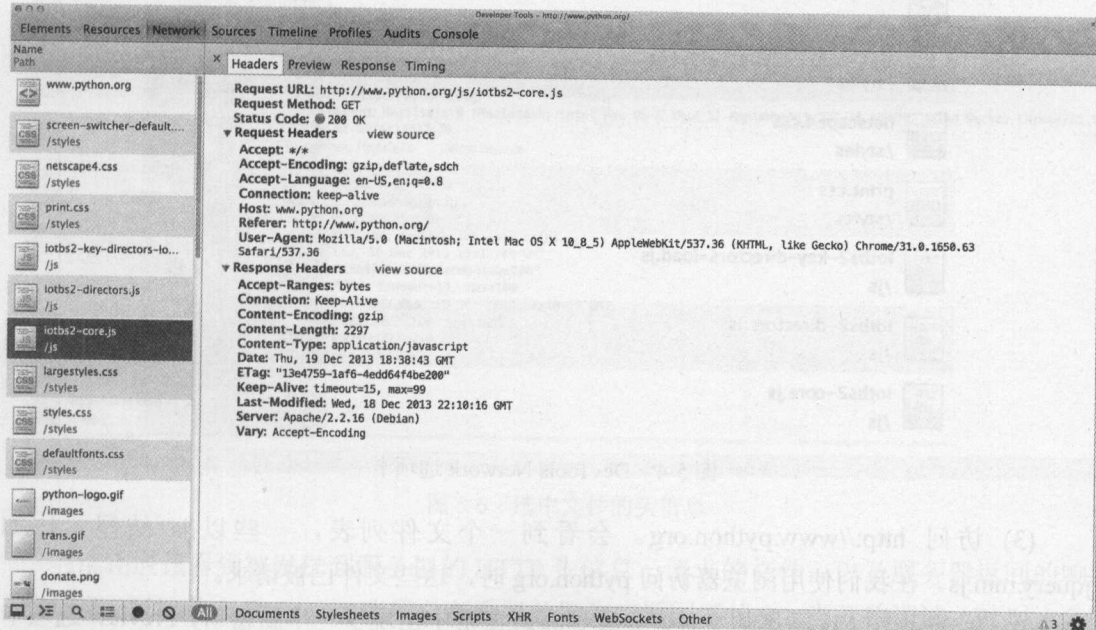


图 5-3 Chrome 开发者工具,展示了向服务器请求文件的头信息

从图 5-3 可以看到，我们访问了 `http://www.python.org`。完成该操作时，我们已经向服务器发送了一个 GET 请求来获得名为 `iotbs2-core.js` 的 JavaScript 文件。服务器响应了“200 OK”状态码。这只是意味着文件可以并已经从服务器获得。还有其他的 HTTP 状态码：最有名的是 404，它代表“找不到”，意思是在服务器上找到文件，所以不能提供。

头包含服务器所需要的信息。服务器通过头知道你在请求什么。这帮助 HTTP 服务器或 Web 服务器对 Internet 请求做出正确的响应。头也包含元数据，比如使用什么浏览器发送请求。从图 5-3 可以看到，请求是从运行 OS X 10.8.5 的 Macintosh 的 Chrome 浏览器发送的。

如果将要进行 Web 开发但是没有成熟的前端团队来检查你的 API 实现，那么需要自己调试。了解如何访问你的 API，然后模拟这些访问来测试和调试也是很有帮助的。为此，可以使用 Chrome Developer Tools(简称为 DevTools)。

试一试：使用 Chrome Developer Tools

在这个“试一试”中，将练习使用 Chrome Developer Tools。在本章会使用这个工具。

- (1) 打开 Chrome 浏览器。在菜单中，单击 View 然后选择 Developer。最后，单击 Developer Tools。
- (2) 在 DevTools 窗口，单击 Network 选项卡(如图 5-4 所示)。

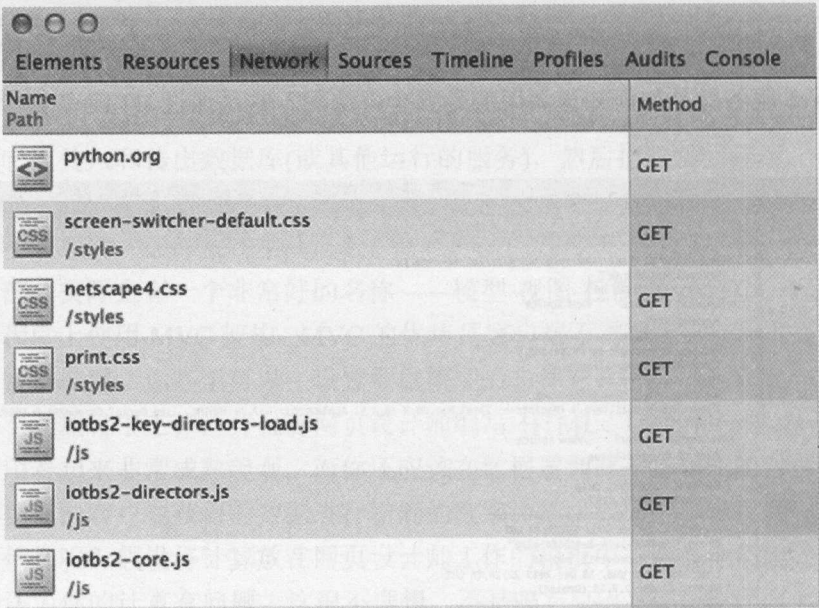


图 5-4 DevTools Network 选项卡

(3) 访问 `http://www.python.org`。会看到一个文件列表，一些以 .js 结尾，比如 `jquery.min.js`。在我们使用浏览器访问 `python.org` 时，这些文件已被请求。

(4) 单击一个以 .js 结尾的文件。会看到右边窗格中出现了一个新窗口，Preview 选项卡高亮显示(如图 5-5 所示)。

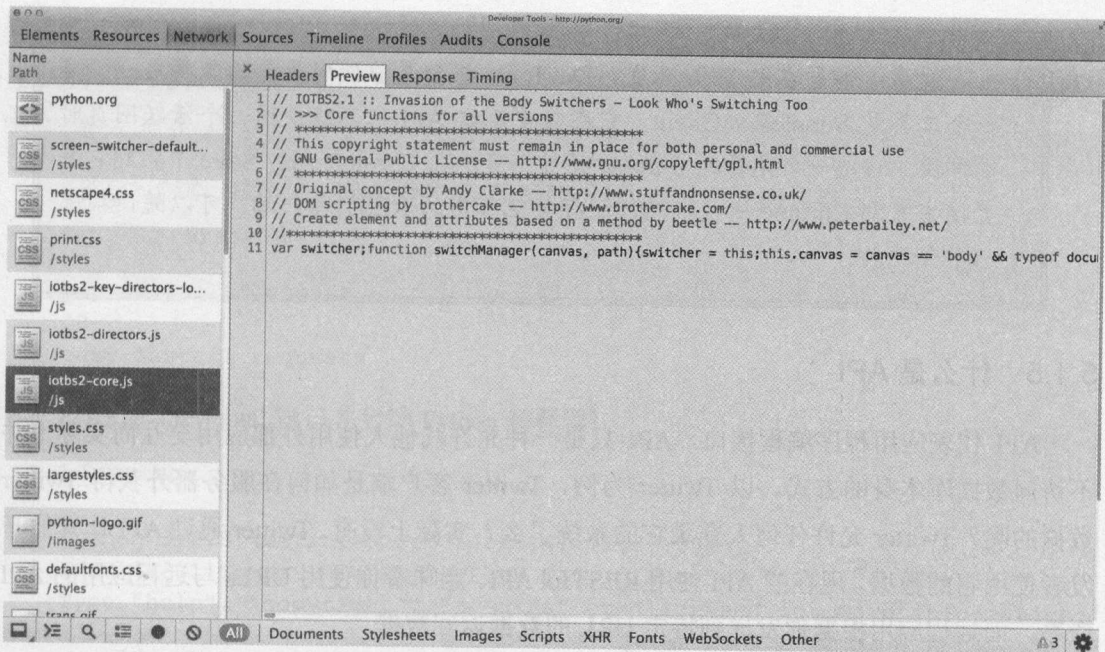


图 5-5 预览选中的文件

(5) 单击高亮的 Preview 选项卡左侧的 Headers 选项卡(如图 5-6 所示)。

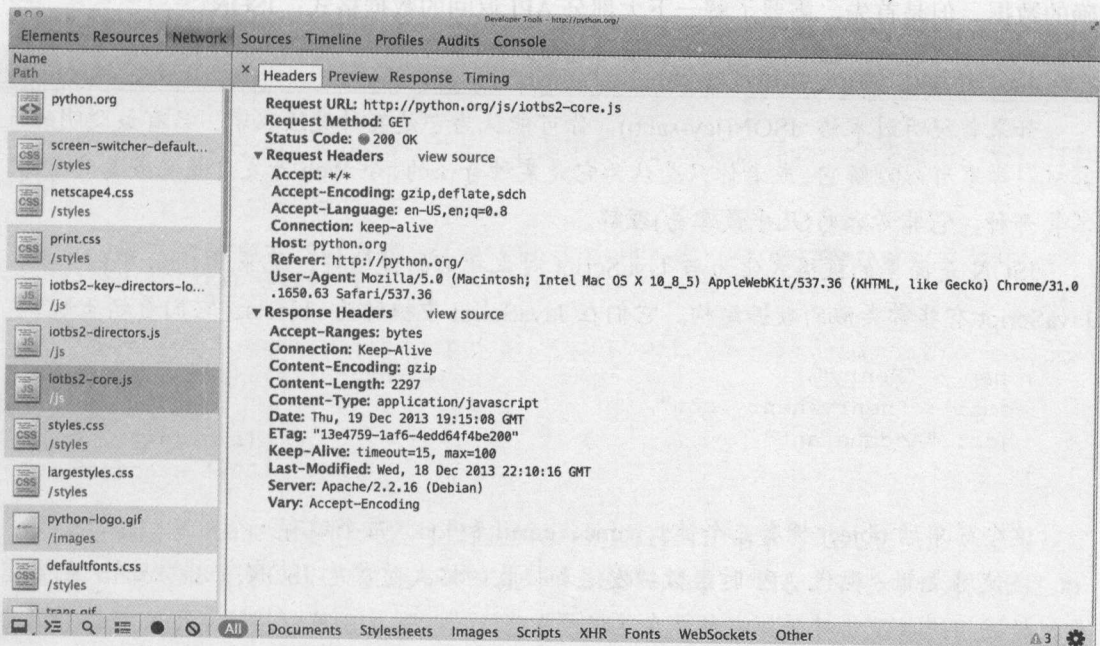


图 5-6 选中文件的头信息

现在应该看到被发送到服务器的 HTTP 头信息、请求的文件，以及服务器返回的响应头信息。可以单击 Response 选项卡查看服务器到底返回了什么。当开始调试 API 时，会更更多地使用 Response 选项卡。



注意：如果在做 Web 开发，DevTools 将会是你使用最多的工具之一。另一个工具是 SimpleRestClient，它在需要测试 API 但不想使用一个活跃网页时很方便。它是一个免费的 Chrome 插件。可以在 Chrome App Store 中找到它并直接安装到 Chrome 中。当然，还有其他可用的 REST 客户端，可以随便挑选一个合适的。

5.1.5 什么是 API

API 代表应用程序编程接口。API 只是一种允许其他人使用外部应用交互而实际上并不访问数据库本身的方式。以 Twitter 为例，Twitter 客户端是如何在服务器外获得 Twitter 数据的呢？Twitter 允许任何人登录它的系统了么？实际上没有。Twitter 通过 API 让其他开发者使用它的数据。现在的 API 使用 RESTful API。意味着你使用 URLs 与远程应用的 API 联系，而远程应用根据你发送到特定 URL 的数据返回数据。

你能发现许多在线教程。它们说明了如何在 Python 中使用一些库访问 API 并拉取数据。接下来，你会创建自己的 API 来提供数据。然后会使用 API 编写客户端文件代码并获得正确的数据。但是首先，需要了解一下大部分 API 返回的数据格式：JSON。

JSON(JavaScript 对象表示法)

如果曾经听过术语 JSON(jay-sahn)，你可能认为它是某种魔法秘密。只有最聪明的计算机科学家可以理解它。或者你只是认为它是某种奇怪的、令 Web 开发者趋之若鹜的东西。不管哪种，它非常容易(几乎最容易)理解。

JSON 是简单的被格式化为与 JavaScript 对象相似的键值对。就像 Python 中的字典，JavaScript 有非常类似的数据结构。它们在 JavaScript 中被称为 objects。它们看起来如下：

```
{ name: "Henry",  
  email: "henry@henry.com",  
  job: "Accountant"  
}
```

这个简单的 object 拥有三个键：name、email 和 job。每个键有一个对应的值。

当使用大部分现代 API 时，数据发送和接收的格式通常是 JSON。这意味着我们只需要把数据组织成键值对并把它包裹在花括号中({})。一些 API 可以接收 JSON 数据并返回 JSON 数据。而一些 API 只是提供 JSON 数据。我们会了解一个发送 JSON 数据的第三方 API。将介绍如何解析数据、如何读取数据，以及 Python 如何处理这些数据。

在接下来的“试一试”中，将学习如何访问第三方 API 来获取数据。在这个示例中，你会使用 USDA 的 API 来获得某一给定区域的农贸市场的数据。可以在 www.data.gov 上

找到很多其他的政府 API。可以在 <http://search.ams.usda.gov/farmersmarkets/v1/svcdesc.html> 上找到关于这个示例的文档。

试一试：使用第三方 API

在这个“试一试”中，会找到给定 U.S.ZIP 码的农贸市场的列表。

(1) 安装 requests 库：

```
pip install requests
OR
easy_install requests
```

(2) 打开 Terminal 窗口并启动 Python 解释器：

```
$ python
```

```
Python 3.3.3 (default, Feb 14 2014, 12:35:03)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

(3) 导入 requests 库。如果 requests 安装正确，应该会看到在按 Return 或 Enter 键之后会立刻返回提示符。如果没有，则说明 requests 没有被正确安装。在 requests 网站查看文档，看看如何修复它。

```
>>> import requests
>>>
```

(4) 向 USDA 的 API 发送一个请求。把请求返回的对象保存到一个变量中，然后打印结果：

```
>>> url = requests.get("http://search.ams.usda.gov/farmersmarkets/v1/data.svc/zipSearch?zip=46201")
```

```
// Feel free to replace the zip code '46201' with your own zip code to see local farmer's markets in your area
```

```
>>> print(url)
<Response [200]>
```

这里调用了 requests.get()。之前我们已经讨论过 HTTP 请求方法。它向服务器发送了一个 GET 请求来获得一些数据(在本例中是一个 JSON 对象)。关于 HTTP 响应状态码 200，我们知道什么信息呢？这意味着收到请求并返回 OK，意思是所有事情都 OK。

(5) 为对象调用 json() 方法，这样它变成了 Python 可以理解的形式。保存对返回的对象调用 json() 得到的结果并打印变量：

```
>>> results = url.json()
```

```
>>> print(results)
```

```
{'results': [{'id': '1003905', 'marketname': '1.7 Irvington Farmers Market & Art Fair'}, {'id': '1005421', 'marketname': '2.4 Original Farmers' Market at Indianapolis City Market'}, {'id': '1006165', 'marketname': '2.4 Indy Winter Farmers Market'}, {'id': '1002600', 'marketname': '2.9 Statehouse Market'}, {'id': '20312', 'marketname': '2.9 Stadium Village Farmers Market'}, {'id': '1002467', 'marketname': '3.8 Farmers Market at the Barn'}, {'id': '1004011', 'marketname': '4.2 Hurlock Farmers' and Watermen's Market'}, {'id': '1003161', 'marketname': '4.4 38th & Meridian Farmers Market'}, {'id': '1006494', 'marketname': '6.1 52 & Shadeland Avenue Farmers Market'}, {'id': '1005505', 'marketname': '6.6 Broad Ripple Farmers Market'}, {'id': '1006523', 'marketname': '6.7 Binford Farmers Market'}, {'id': '1003901', 'marketname': '7.9 Cumberland Farmers Market'}, {'id': '1004663', 'marketname': '12.1 Geist Farmers Market'}, {'id': '1009026', 'marketname': '12.1 Farm to Fork Farmers Market at Normandy Farms'}, {'id': '1007367', 'marketname': '13.5 Carmel Farmers Market'}, {'id': '1000748', 'marketname': '13.6 Fishers Farmers Market'}, {'id': '1004686', 'marketname': '14.5 Zionsville Farmers Market'}, {'id': '1008028', 'marketname': '16.0 Brownsburg Farmers Market'}, {'id': '1000129', 'marketname': '16.2 Plainfield Chamber Farmers' Market'}]}
```

当查询 ZIP 码后你所看到的是所有市场的列表。这个请求得到的数据位于 USDA 的服务器上。然而，因为 USDA 已经提供了 API，所以可以在不真正访问数据库本身的情况下访问这个数据。

返回的数据类型是什么？它是一个只有一个元素的字典，它的键是 `results`，包含了一个字典列表作为它的值。你怎么知道这些的？`{ = dict indicator..... [= list indicator..... { = dict indicator`。第一个字典项被命名为 `results`，而它的数据是一个列表。切记，字典是键值对，而列表只是数据索引(从 0 开始索引)。所以，如何将列表从第一个字典中取出来呢？必须使用键来引用字典——在本示例中，键就是 `results`。

(6) 为了得到那个键的值，可以通过如下代码调用它：

```
>>> for result in results['results']:
...     print(result)
{'id': '1003905', 'marketname': '1.7 Irvington Farmers Market & Art Fair'}
{'id': '1005421', 'marketname': '2.4 Original Farmers' Market at Indianapolis City Market"}
{'id': '1006165', 'marketname': '2.4 Indy Winter Farmers Market'}
{'id': '1002600', 'marketname': '2.9 Statehouse Market'}
{'id': '20312', 'marketname': '2.9 Stadium Village Farmers Market '}
{'id': '1002467', 'marketname': '3.8 Farmers Market at the Barn'}
{'id': '1004011', 'marketname': '4.2 Hurlock Farmers' and Watermen's Market"}
{'id': '1003161', 'marketname': '4.4 38th & Meridian Farmers Market'}
{'id': '1006494', 'marketname': '6.1 52 & Shadeland Avenue Farmers Market'}
{'id': '1005505', 'marketname': '6.6 Broad Ripple Farmers Market'}
{'id': '1006523', 'marketname': '6.7 Binford Farmers Market'}
{'id': '1003901', 'marketname': '7.9 Cumberland Farmers Market'}
{'id': '1004663', 'marketname': '12.1 Geist Farmers Market'}
```

```
{'id': '1009026', 'marketname': '12.1 Farm to Fork Farmers Market at Normandy Farms'}
{'id': '1007367', 'marketname': '13.5 Carmel Farmers Market'}
{'id': '1000748', 'marketname': '13.6 Fishers Farmers Market'}
{'id': '1004686', 'marketname': '14.5 Zionsville Farmers Market'}
{'id': '1008028', 'marketname': '16.0 Brownsburg Farmers Market'}
{'id': '1000129', 'marketname': '16.2 Plainfield Chamber Farmers' Market'}
```

当运行这段代码时, 应该看到一个 Unicode 编码的 ZIP 码 46201(或你的 ZIP 码, 如果修改了数据)之内的 Indianapolis 区域的农贸市场列表。

示例说明

基于 USDA 数据库中的数据, 刚刚使用了 USDA 的 API 搜索了特定 ZIP 码内的农贸市场。从服务器得到了数据并使用 requests 库的 json() 方法。你从服务器发回的 JSON 格式的数据中取出了那个信息, 这样就可以读取它。然后注意到了数据结构, 这样可以获得每个单独列表并通过一个 for 循环列出它们。

如果想获得农贸市场的名称, 该怎么办呢? 用 Google 搜索它吗?

幸运的是, USDA 已经做了繁重的通过 Google 地图查找每个市场的工作。它为每个市场提供了对应的 Google 地图链接。根据 USDA 的 API 文档(<http://search.ams.usda.gov/farmersmarkets/v1/svcdesc.html>), 如果传入一个之前查询中获得的 market ID, 这个搜索会返回一些有关该市场的详情。结果中包含了一个 Google 地图的链接。如下:

```
>>> market = "http://search.ams.usda.gov/farmersmarkets/v1/data.svc/mktDetail?id="

>>> for result in results['results']:
...     id = result['id']
...     details = requests.get(market + id).json()
...     print(details['marketdetails']['GoogleLink'])
http://maps.google.com/?q=39.7776%2C%20-86.0782%20(%22Irvington+Farmers+Market+%26+Art+Fair%22)
This is the example of one of the links that may be returned.
```

这段代码的作用是什么? 你能破译它么? 试着用笔和纸把它以人类可读的格式写出来, 或者打出来, 就好像你正在跟别人解释代码。如果不确定到底发生了什么, 在每一行后插入打印语句, 看看每行代码是什么样的。如下:

```
>>>for result in results['results']:
...     id = result['id']
...     print(id) [RS - all these need parenthecsss[
...     print result['id']
...     details = requests.get(market + id).json()
...     print details
...     print details['marketdetails']
...     print details['marketdetails']['GoogleLink']
```


作为一个趣味练习，打开 Chrome 并访问 URL: <http://search.ams.usda.gov/farmersmarkets/v1/data.svc/zipSearch?zip=46201>。注意你得到的结果。打开 DevTools，看一下响应头信息(如图 5-7 所示)。注意，这个响应与你在 Python 解释器中请求得到的响应(200 OK)是相同的。接下来，单击 Preview 选项卡。你应该看到每一个 JSON 对象的返回结果。可以随便单击其他的选项卡，查看浏览器在你查看的数据之外还显示了什么数据。

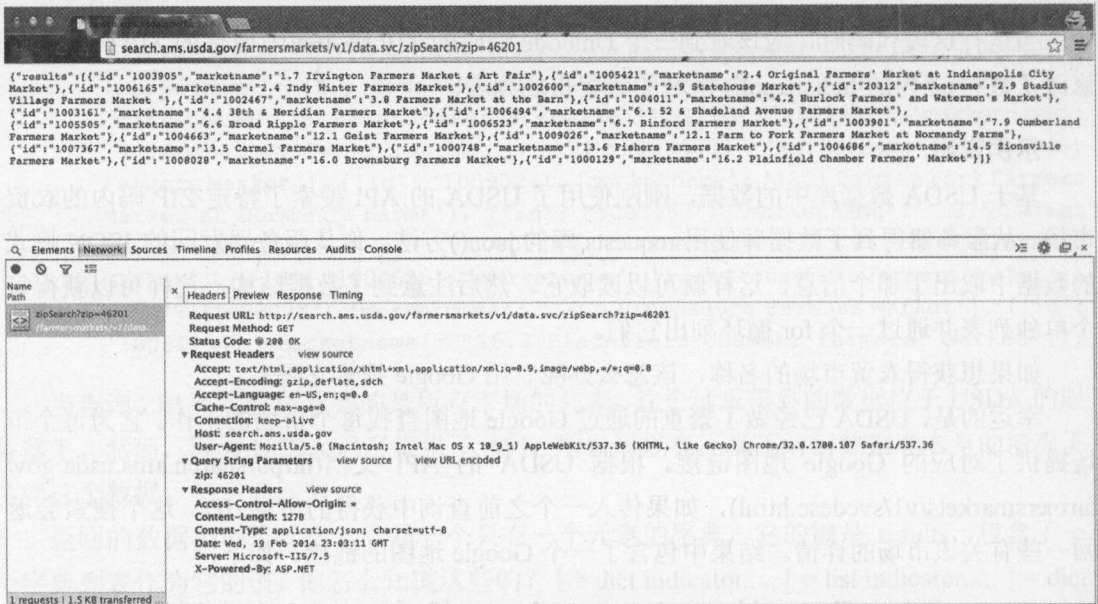


图 5-7：Chrome DevTools 的头信息标签，它展示了 USDA 网站的 HTTP 响应

所以，现在的问题是：工作原理是什么？Requests 库是如何工作的？这是我们下一节要介绍的内容：使用 Python 进行 Web 编程。尽管我们不会创建自己的 requests-like 库，但会介绍为了实现这样一个库，Python 所提供的底层逻辑和功能。

5.2 使用 Python 进行 Web 编程

在上一节中，你了解了 Requests 库。本节将介绍少许 Requests 在底层使用的技术。然后，使用 FlaskWeb 框架(<http://flask.pocoo.org>)，为借出库增加一个 Web 界面，并且通过浏览器让它变成可交互的。但是首先，在为之付出更多精力之前，应该理解它们的工作原理。

使用 Python HTTP 模块

Python 非常强大、用途广泛且易用。其中一部分原因是基础 Python 安装自带了很多内置功能。当安装 Python 时，你也得到了很多可以使用的模块。这些模块中就包括一些 http 模块：`http.server`、`http.client` 和一些其他模块。在本节中将使用 `http.server` 模块。你会发现使用它来启动一个快速的 HTTP 服务器十分轻松。这样可以在调试或测试时快速地启动网

页。本节也演示了一些更加常用的第三方库如何使用 Python 的内置模块为 Python 开发者创建非常强大的工具。

下面首先介绍 `http.server` 模块，看看可以多么快速地在本地启动和运行一个 HTTP 服务器。

1. 创建 HTTP 服务器

在这个“试一试”中，将使用几行代码设置一个 HTTP 服务器。它会在本地运行并提供那些你在本地目录创建的网页。在这个练习之后，你就能很好地理解一些更加常用的框架和第三方库如何实现 Python 内置模块利用强大的力量来创建好用的工具。

试一试：通过 `http.server` 提供本地文件

这个“试一试”会演示如何通过 Web 浏览器提供本地目录中的文件，以及测试或调试一个 Web 服务器运行所需的任何代码。它也演示了 Python 的内置模块的强大功能。

(1) 在项目目录中，为第 5 章创建一个目录。使用自选的编辑器创建 `index.html` 文件并保存。文件包含如下代码：

```
<!doctype html>
<html>
  <head>
    <title>HELLO WORLD!</title>
  </head>
  <body>
    <p> Hello World! I am serving this page via Python! WOWZERS!</p>
  </body>
</html>
```

(2) 打开一个新的 Terminal 窗口并启动 Python 解释器。导入两个所需的内置模块：

```
Python'
Python 3.3.3 (default, Feb 14 2014, 12:35:03)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import socketserver
>>> import http.server
```

(3) 决定要通过哪个端口提供数据。可以随意选择端口，但这个示例使用了 8080：

```
>>> PORT = 8080
```

注意，变量 `PORT` 是全天大写的，因为它是常量。这意味着它在程序中不会改变。

(4) 设置一个名为 `Handsy` 的处理程序来处理将会发送的请求。

```
>>> Handsy = http.server.SimpleHTTPRequestHandler
```

(5) 既然已经有了一个请求处理程序，就可以设置简单的 HTTP 守护程序(或 `httpd`，因为这是 UNIX 命名的惯例)：

```
>>>httpd = socketserver.TCPServer(("", PORT), Handsy )
```

在此传入了一个空字符串、PORT 常量，以及作为请求处理对象变量的 Handsy。可以用通俗的语言讲述将要发生什么吗？可以在纸上用段落的形式写下来，这会有所帮助。

(6) 最后，需要将目录提供给正确的端口，这样可以通过浏览器访问它：

```
>>> httpd.serve_forever()
```

(7) 让 Terminal 窗口开着并一直保持运行状态，打开浏览器并访问 `Http://localhost:8000/`。你应该看到类似图 5-8 所示的网页。

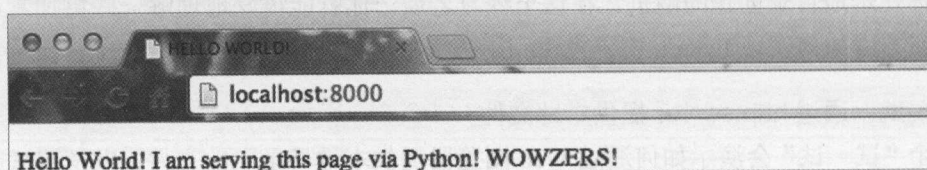


图 5-8 注意它是如何显示之前创建的 HTML 文件的

(8) 回到服务器运行的 Terminal 窗口。应该看到如下内容：

```
>>> httpd.serve_forever()
127.0.0.1 - - [14/Feb/2014 14:17:02] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [14/Feb/2014 14:17:04] "GET / HTTP/1.1" 200 -
```

还记得状态码 200 是什么意思吗？OK 意味着服务器找到了 `index.html`，并且可以把它提供给客户端。很简单吧，不是吗？

示例说明

Python 的内置 HTTP 模块有助于你快速搭建一个小的 HTTP 服务器来提供简单的网页。`http.server` 模块会在提供的目录(你提供了`"`，它表示启动服务器时所处的目录)中搜索 `index.html` 文件。如果 `index.html` 文件存在，它会提供这个文件。如果文件不存在，`http.server` 会使用 `list_directory()` 方法列出当前目录的内容。这对调试和测试代码非常有帮助。但是它不应该被用于生产系统中。然而，如果想要创建一个更大的库，比如你将会使用的 Flask 库或之前使用的 Requests 库，可以围绕这些内置模块编写代码，以后可以使用这些内置模块创建一些非常有用的工具。



注意：Python 3 让创建一个 HTTP 服务器更加容易。可以进入想要通过 HTTP 服务器提供的目录，运行这个命令：

```
python -m http.server 8000.
```

这个命令会在 `localhost` 上通过你提供的端口提供目录。所以，如果进入 `index.html` 文件所在的 `Chapter5` 目录并在终端运行这个示例的话，可以在浏览器中打开 `http://localhost:8000` 并看到 `index.html` 文件！在一个没有 `index.html` 文件的文件夹中试试，看看会发生什么结果。

2. 探索 Flask 框架

现在，你会更加细致地了解 Flask 框架。它是一个用 Python 编写的第三方库，可以帮助你做很多事情，从创建你自己的 API 到创建一个完整的基于 Web 的应用。Flask 支持模板，使用的是 Jinja 模板系统。因此，可以把它作为一个多合一的 Web 框架。你会在本章的下一节中使用这个功能。

在开始之前，需要先安装 Flask(`pip install flask`)。你会使用 SQLite3 数据库。幸运的是，Python 3 自带了 SQLite3。SQLite 是一个较小的数据库，它特别适合这个简单的示例应用。如果在做一个较大的应用，我们推荐你花一些时间来研究更大的数据库，比如 PostgreSQL、SQLAlchemy 或许多其他可用的数据库，还包括 NoSQL 数据库，比如 Redis。然而，对于本章的目的来说，你会使用 SQLite。它是一个非常易用、小巧而且轻量级的数据库。

安装好 Flask 之后，首先创建一个非常简单的应用，之后将其构建为一个更大的应用。



注意：Python 3 支持 Flask，但仅在 3.3 或更新版本中支持。Flask 不能工作在 3.2 和旧的版本上。同样，大部分插件都是在版本 2.7 下开发的。所以把它们移植到 Python 3，尽管可能，但要留给读者去实现。当遇到任何异常并且不能在 Internet 上搜到帮助时，回到 Python 2.7，推荐在该环境下使用 Flask。

首先，想要设置项目的每个部分所处的目录结构。可以通过多种方式设置，而每个教程都会为你演示不同的方式，但基本思想是一致的。因为会在一个类 MVC 的环境中工作，所以想要分隔项目的每个部分。在接下来的“试一试”中，在解压缩 `lendy.zip` 文件后，应该拥有类似如下的目录结构：

```
lendy (借出应用的名称)
```

```
static
```

```
templates
```

这和大多数项目的初始结构类似。想要分隔项目的每个部分。`static` 目录中保存的是静态文件，比如图片和脚本。`templates` 目录中保存的是模板。

下面介绍一个典型 MVC 架构的各个部分：

- **模型：**这是保存数据模型或数据库表的地方。它是用户所建的表(不安全)以及你将借出的物品。
- **视图：**用户将会看到在浏览器中为他们所展示的内容，或如何与数据进行交互，以及在控制器完成工作后数据的去向。

- **控制器**：这是应用中控制数据流的部分。用户与这部分进行交互。所以，一个用户可能向后端发送表单数据，这是通过项目的控制器部分得到的。这部分与模型交互，并且执行更新、创建甚至是删除操作。这也是应用中最繁重的部分，因为这部分从模型层接收数据并把它返回到视图中。从根本上讲，这是 Web 应用的大脑。

下面通过创建 Web 应用的基础部分，来介绍这些部分是如何协同工作的。

3. 在 Flask 中创建数据模型

在这个示例中，将使用在第 3 章“创建 LendyDB SQL 数据库”部分创建的 lendydb 数据库。使用两个数据实体：成员和物品。成员将可以登录并查看物品的库存，并向库存中添加物品。你将会在这个练习中使用文本编辑器并保存文件。由于时间的关系，你不会使用任何安全措施。然而，在创建公共 Web 接口前应该对最常用的安全实践非常熟悉。

这个应用的数据模型包含第 3 章的两个文件：API 文件 `lendydata.py` 和数据库文件 `lendy.db`。需要将这两个文件复制到 `lendy` 文件夹中。

4. 创建核心 Flask 文件

现在是时候真正接触应用的核心了，也就是那些为我们提供功能并允许运行 Web 应用的所有东西。下面为应用创建 Python 文件。

试一试：创建一个简单的 Flask 应用(lendy.zip)

这个“试一试”将演示如何开始一个 Flask 项目。

(1) 在项目目录(`lendy`)中，打开文件 `lendy.py`。它包含以下内容：

```
import sqlite3, os, lendydata
from flask import Flask, request, session, g, redirect, url_for, abort, \
    render_template, flash
```

前两行导入了将使用的库和模块。第一行导入了 `sqlite3` 和 `os` 模块。这些是 Python 自带的模块。这一行还导入了在第 3 章创建的 `lendydata` API。第二行只导入了 Flask 库中的模块。在导入模块时，这是很好的做法。只导入那些实际使用的模块和库。记住，在导入时，实际是在运行在导入中指定的文件的所有代码。所以，如果一个模块有 40 个方法，将会导入所有这些方法的定义。所以，只应该导入你会使用的模块。

(2) 配置你的应用：

```
app = Flask(__name__)

# Load default config and override config from an environment variable
app.config.update(dict(
    DATABASE=os.path.join(app.root_path, 'lendy.db'),
    DEBUG=True,
    SECRET_KEY='nickknackpaddywhack',
    USERNAME='admin',
```

```

PASSWORD='thisisterrible'
))
app.config.from_envvar('LENDY_SETTINGS', silent=True)

```

在此创建了一个 Flask 对象，并将其命名为 `app`。然后，设置了 Flask 的配置属性。由于所有东西在 Python 中都是一个 first-class 对象，因此应该在需要时修改这个对象。

注意在前两行传入的 `__name__` 变量。记住，Python 脚本的运行情况依赖于它的执行方式。它可以作为 `__main__` 模块通过命令 `python app.py` 直接执行。也可以在另一个(主)Python 文件中被导入而间接执行。Python 解释器将 `__name__` 变量设置为 `__main__` 或被导入的文件名。在此将 `__name__` 变量传给 Flask 和 `config.from_object` 方法。根据执行脚本的方式，`__name__` 变量为 `__main__` 或脚本名(在本示例中，`__name__` 变量会被设置为 `lendy`)。

接下来，更新了与 Flask 对象一起使用的字典。Python 对象从本质上说都是字典。这也是为什么在任何时候都可以修改它们的原因。全大写意味着设置的是常量(永远不会改变的变量)。这些变量会被传入应用中，并帮助你运行程序。

首先，有一个 DATABASE 变量。它声明了数据库文件的位置。SQLite 创建一个简单文件作为数据库，这也是为什么它被称为 lite 的原因所在。调用 `os.path.join()` 只是设置了之后会使用的 SQLite 文件的路径。

其次，设置了 `DEBUG=True`。这是因为处于开发模式下。当准备好将 Flask 项目放到生产环境时，希望设置它为 `FALSE`。

接下来，使用了 `SECRET_KEY` 变量。这个变量应该是某种随机的、长度很长的且很难猜测的东西。在此，为了说明示例，这个变量并没有这样设置。这个 key 有助于你保证客户端会话的安全。

最后，设置了 `USERNAME` 和 `PASSWORD`。这些是应用的证书。由于创建的是一个非常简单的应用，因此将会把这些存储在配置中而不是数据库中。当然，这些东西应该比在此演示的示例更难以设置，它们应该被加密后放进数据库。但是在此，安全不是最先被考虑的因素。同样，我们也假定应用会保持私有并且只在本地运行。

最后一行对以后很重要。它设置了一个名为 `LENDY_SETTINGS` 的环境变量，其中包含了配置变量。然而，你已经在这里设置了，所以这个配置文件其实并不存在！这也是为什么要设置 `silent=True` 标签的原因。它会忽略配置文件不存在时产生的错误。如果想要增加一个配置文件，则需要将 `LENDY_SETTINGS` 变量指向那个配置文件。

(3) 现在，剖析数据库连接，因为这是应用的核心部分：

```

def get_db():
    """Opens a new database connection if one does not exist for our current
    request
    context (the g object helps with this task)"""

    if not hasattr(g, 'sqlite_db'):
        lendydata.initDB()
        g.sqlite_db = lendydata.db

```



```

    return g.sqlite_db

@app.teardown_appcontext
def close_db(error):
    """Closes the database again at the end of the request. Note the 'g'
    object which makes sure we only operate on the current request."""
    if hasattr(g, 'sqlite_db'):
        lendedydata.closeDB()

```

简言之，这段代码打开了一个数据库连接，这样你就拥有一个 SQLite 数据库句柄，并且可以在上面执行函数。其中有一个 `init_db` 函数，一会儿你也会用到这个函数。在 `get_db()` 方法调用中还有一点 Flask 特性。这个函数在没有连接存在时会打开一个新的连接，而 `g` 变量在 Flask 中是一个特殊对象，它只对激活请求有效。这在不同的请求对象间保持了数据的一致性。最后，代码打开了数据库。然后，在应用使用完连接之后，使用 `lendedydata.close()` 函数关闭了连接。

注意在 `@app.teardown_appcontext` 方法中再一次使用了 `g` 对象。这个对象实际上是 `flask.g`，但是因为你直接导入了它，所以可以简单地用 `g` 即可。这个对象保持每个请求分开。这样在关闭一个数据库连接时，并不会关闭所有数据库连接。这就是使用库的轻松和简洁之处。许多时间库包含类似于这个 `g` 对象的东西。它会让你的工作更加轻松，这样可以只完成需要做的任务而不必关心复杂的东西(在此是每个连接的打开和关闭数据库连接)。

(4) 增加这段代码：

```

if __name__ == '__main__':
    app.run()

```

所有 Python 脚本函数都通过这段代码被调用。还记得我们之前讨论的 `__name__` 和 `__main__` 吗？它们确切的意思是什么？当 Python 执行一个脚本时，会设置一个隐藏变量 `__name__`。当这个脚本作为第一个 Python 脚本执行时，这个 `__name__` 属性被设置为 `__main__`。这样解释器就知道这是第一个脚本。任何在 `__main__` (通过导入语句)之后调用的脚本都会把它们 `__name__` 变量设置为它们的文件名。所以，如果导入了 `os`，那么 `os` 脚本(它只是一个 `.py` 文件)会将它的 `__name__` 变量设置为 `os`。这非常像其他语言中的命名空间，它其实是命名空间在 Python 中实现的基本版本。

(5) 打开 Terminal 并在保存 `lendedy.py` 文件的目录中，运行以下命令：

```
python lendedy.py
```

将看到如下信息：

```

* Running on http://127.0.0.1:5000/
* Restarting with reloader

```

(6) 现在，在 Web 浏览器中访问 `localhost:5000`。你应该在浏览器中获得一个 404 错误，

并且在 Terminal 窗口中会看到如下信息:

```
127.0.0.1 - - [18/Feb/2014 12:10:44] "GET / HTTP/1.1" 404 -
```

(7) 服务器返回一个 404。这是因为你还没有任何视图(或模板)。下面添加它们。在 lendy.py 文件中添加如下代码行:

```
@app.route('/')
@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != app.config['USERNAME']:
            error = 'Invalid username'
        elif request.form['password'] != app.config['PASSWORD']:
            error = 'Invalid password'
        else:
            session['logged_in'] = True
            flash('You were logged in')
            return redirect(url_for('show_inventory'))
    return render_template('login.html', error=error)
```

这才是 Flask 在实际应用中真正强大的地方。注意, Flask 使用了修饰符方法来创建 HTTP 路由。这为你节省了大量时间, 也让代码的维护更简单。通过这些特定的路由, 可以告诉 Flask: 当有人访问/login 路径或根目录(/)时, login 函数就会被调用, 并处理请求方法的值(我们已经在本章前面讨论过这一点。在此, 我们允许使用 POST 和 GET 方法, 并分别处理它们)。然后检查了传入的用户名和密码。如果为真, 就提醒用户他们已经成功登录了, 并通过 render_template()方法将其带入 login.html 模板。从本质上说, 希望用户访问网站的登录页面。

(8) 为其他的页面设置其他视图。在 lendy.py 文件中添加如下代码:

```
@app.route('/inventory')
def show_inventory():
    get_db()
    allItems = lendydata.get_items()
    inventory = [dict(zip(['name', 'description'], [item[1], item[2]]))
                  for item in allItems]
    return render_template('items.html', items=inventory)
```

这段代码与之前的类似。当用户访问/inventory 端点时, show_inventory 方法就会被调用。这个函数会调用前面的 get_db()函数, 然后执行 get_items() API 函数, 查询表中所有的物品。接下来创建了一个在网站中使用的字段名和值的字典, 将这个字典传入模板中并填充内容。注意, 这个端点在 route()方法中并没有 methods=参数, 这是因为默认的方法是 GET, 所以不需要传递这个参数。

(9) 如果想要向库存中添加物品, 该怎么办呢? 下面向 lendy.py 文件中添加这个端点:

```

@app.route('/add', methods=['POST'])
def add_item():
    if not session.get('logged_in'):
        abort(401)
    get_db()
    ownerID = [row[0] for row in lendydata.get_members()
               if row[1] == request.form['owner']]
    try: ownerID = ownerID[0]
    except IndexError:
        # implies no owners match name
        # should raise error/create new member
        ownerID = 1      # use default member for now.

    lendydata.insert_item(request.form['name'],
                           request.form['description'],
                           ownerID,
                           request.form['price'],
                           request.form['condition'])

    flash('New entry was successfully posted')
    return redirect(url_for('show_inventory'))

```

这个方法使用 `get_members()` API 函数根据表单获得的名称来查找拥有者的 ID。然后调用 `add_item()` API 函数将物品添加到数据库中。

你仍然看不到任何东西，因为你实际上还没有创建模板。模板是浏览器实际展示给用户的東西。这也是 HTML 的魔力所在。但真正的魔力在于让那些模板可以向 Python 中间件传送数据。下面介绍如何实现这一点。

接下来，你将了解模板的知识。

可以在 `lendy/templates` 文件夹中的 `lendy.zip` 文件中找到模板。我们只会简要地讨论每个文件，让你对所有东西如何交互有个概念。

(10) 在 `templates` 目录中打开 `base.html` 文件并查看它的内容：

```

<!doctype html>
<title>Inventory Of Things</title>
<link rel=stylesheet type=text/css href="{{ url_for('static',
filename='style.css') }}">
<div class=page>
    <h1>Lendy</h1>
    <div class=metanav>
        {% if not session.logged_in %}
            <a href="{{ url_for('login') }}">log in</a>
        {% endif %}
    </div>
    {% for message in get_flashed_messages() %}
        <div class=flash>{{ message }}</div>
    {% endfor %}

```



```
{% block body %}{% endblock %}
</div>
```

在处理模板时，通常有一个基础的 HTML 文件，它将保存网站的基本架构。这通常包括导航和页面设置的其他部分，这些部分在应用中不会发生变化。你可能喜欢很多只有一页的 Web 应用，它们的工作原理就是使用模板。

在有了基础模板之后，就可以开始创建其他页面了。需要一个登录页面、一个库存列表页面和一个库存添加页面。下面介绍每个页面的重要部分，这样能够更好地理解这些部分是如何协同工作的。

(11) 在 lendy 应用的 templates 文件夹中，创建一个 login.html 文件并添加以下代码来创建登录页面：

```
{% extends "base.html" %}
{% block body %}
<h2>Login</h2>
{% if error %}<p class=error><strong>Error:</strong> {{ error }}{% endif %}
<form action="{{ url_for('login') }}" method=post>
  <dl>
    <dt>Username:
    <dd><input type=text name=username>
    <dt>Password:
    <dd><input type=password name=password>
    <dd><input type=submit value=Login>
  </dl>
</form>
{% endblock %}
```

注意第一行表示这是一个从 base.html 扩展而来的模板。在最开始就声明它，这样模板引擎就会明白你想要向页面导入 base.html 的所有部分。每页都会包含这一行，表示页面会从哪个 HTML 文件扩展。

其次，注意 {% block body %} 行。这意味着你有一个主体块(body block)，它会包含你想要显示的 HTML。所有 HTML 都被包含在 {% block body %} ... {% endblock %} 符号中。这些标签表示模板的动态部分。比如，下面这行代码：

```
{% if error %}<p class=error><strong>Error:</strong> {{ error }}{% endif %}
```

注意 {% if error %} 标签和 {{ error }} 标签之间的区别。双花括号({{...}})表示将使用应用中的数据填充空白处。括号内部的值是将通过 Flask 代码中的函数传给模板的变量。带有百分号的花括号({%...%})表示模板引擎将执行的一个动作。所以 {% blockbody %} 向模板引擎表示你在开始代码块的主体部分，而 {% if error %} 标签表示你将在模板中执行一些逻辑。这个符号在许多模板引擎中非常常见，所以可能会经常看到它。

(12) 在浏览器中重载该应用。当访问 <http://localhost:5000> 或 <http://localhost:5000/login> 时，应该看到一个登录页面。如果发生错误，请再次检查代码输入和缩进。同时，回想一

下本章的第一个“试一试”部分，并使用 Chrome DevTools 来检查请求中的任何错误。

现在想要添加一些用来列出物品的页面并将物品添加到库存。同样，我们已经提供了这些页面，但是也可以自己把这些代码打印出来。我们只需要了解模板的重要部分。

(13) 打开 `items.html` 并注意 `{%}` 和 `{{}}` 的使用。基于从 `login.html` 模板学到的知识，你能解释发生了什么吗？

```
{% extends "base.html" %}
{% block body %}
    {% if session.logged_in %}
        <form action="{{ url_for('add_item') }}" method=post class=add-item>
            <dl>
                <dt>Item Name:
                <dd><input type=text size=30 name=name>
                <dt>Item Description:
                <dd><textarea name=description rows=5 cols=40></textarea>
                <dt>Item Condition:
                <dd><input type=text size=30 name=condition>
                <dt>Item Price:
                <dd><input type=text size=30 name=price>
                <dt>Owner Name:
                <dd><input type=text size=30 name=owner> <dd><input type=submit
value=Submit>
            </dl>
        </form>
    {% endif %}
    <ul class=entries>
    {% for item in items %}
        <li class="item_list">
            <h2>{{ item.name }}</h2>{{ item.description|safe }}

    {% else %}
        <li><em>There doesn't seem to be anything here. Add some items, maybe?</em>
    {% endfor %}
    </ul>
{% endblock %}
```

在此传递了一个表单动作和方法。如果已经登录，可以设置表单来接收新物品并创建一个 POST 方法。这个方法随后会传给 `'add_item'` URL，这是到 `lendy.py` 文件中 `add_item` 函数的一个映射。

这个模板中另一个有意思的部分是它在一个模板中包含了两个功能。第一个功能是添加物品，而另一个是列出物品。如果已经登录，可以添加物品并查看物品列表。如果没有，就只能看到物品列表。你有很多方式可以显示这个信息。可以为列出物品和添加物品创建单独的页面。可以使用 CSS 隐藏 divs 并且只有在用户单击按钮时才显示添加的物品部分。可以随便试试这个功能的其他选项，找到你最喜欢的一个。现在我们使用一种快速且粗暴的方式。

下一步是在添加 CSS。这一步非常简单。我们已经在 `static` 文件夹中提供了 CSS 文件 `style.css`。在一个项目中，`style.css` 通常作为基础 CSS 文件。我们的 CSS 非常简单，但其作用很强大，而它的功能非常神奇。可以随意使用不同的 CSS 值来实现你的应用。

(14) 打开 `static` 文件夹中的 `style.css` 文件并查看：

```
body{
    font-family: sans-serif;
    background: #eee;
}
a, h1, h2{
    color: #377ba8;
}
h1, h2{
    font-family: 'Georgia', serif;
    margin: 0;
}
h1{
    border-bottom: 2px solid #eee;
}
h2{
    font-size: 1.2em;
}
.page{
    margin: 2em auto;
    width: 35em;
    border: 5px solid #ccc;
    padding: 0.8em;
}
.inventory{
    list-style: none;
    margin: 0;
    padding: 0;
}
.inventory li{
    margin: 0.8em 1.2em;
}
.inventory li h2{
    margin-left: -1em;
}
.add-item{
```



```

        font-size: 0.9em;
        border-bottom: 1px solid #ccc;
    }

    .add-item dl{
        font-weight: bold;
    }

    .metanav{
        text-align: right;
        font-size: 0.8em;
        padding: 0.3em;
        margin-bottom: 1em;
        background: #fafafa;
    }

    .flash{
        background: #cee5F5;
        padding: 0.5em;
        border: 1px solid #aacbe2;
    }

    .error{
        background: #f0d6d6;
        padding: 0.5em;
    }

```

注意，为基础元素(body, h1, h2 等)定义了一些基础风格。然后，定义了一些类(那些以点开始的行，点在 CSS 中表示类，而符号#表示 ID 名称)。

需要特别重视的是 flash 类，这是 Flask 的一个特性。Flask 通过 flash 在页面向用户显示错误消息或其他系统反馈的文本。可以回头看看之前的代码，看看 flash 函数在哪里被调用以及使用它的方式。

至此，已经创建了一个小型的 Flask 应用，可以通过它来追踪库存。

(15) 像之前(获得 404 错误时)那样启动 Flask 应用并查看你现在是否有一个工作中的 Web 应用。

可能会遇到一些错误，所以检查代码的拼写和语法错误(在 Python 中，特别是缩进错误)。如果仍然不清楚是何处导致了错误，则启动 Chrome DevTools 开始探测你发送的请求以及从服务器获得的返回，这可能会有所帮助。记住查看 Network 标签。可能需要刷新页面来用产生的请求填充 Network 标签。

示例说明

刚刚已经使用 Flask 框架创建了一个小型 Web 应用来处理繁重的 HTTP 事务。该应用创建了一个数据库并允许你连接到数据库上。可以在该数据库中创建表来存储数据。然后设置了视图，告诉应用如何和何时返回数据，以及返回什么数据。设置了端点(URL 中的

/<foo>部分), 并将其映射到函数上, 这样有助于从模板中取出数据并发送到数据库中。然后创建了模板来显示所有信息, 使用 Jinja 作为你的模板引擎。之后对模板进行了一些设置, 这样模板就可以与 `lendy.py` 文件来回传递数据。最后还设置了元素的风格, 这样你的应用在用户(就是我们!)看来更漂亮、也更易用。

5.3 有关 Python 和 Web 的更多知识

使用 Python 创建 Web 应用极其容易。你可以自己完成所有的脏活累活, 但也可以使用任意一个开源框架帮助你做所有或部分这些事情。通过 Python, 可以用两种方式创建网站: 静态网站生成器和全功能 Web 框架。本节将简要描述这两种方法, 还包括一些更加流行的生成器和框架。

5.3.1 静态网站生成器

静态网站生成器通常被用于像博客和其他文档类的东西。这时, 你可能想要在某一刻生成一个页面并向外提供这个页面, 而页面的内容在发布之后就不能再改变。一些更流行的静态网站生成器包括:

- **Pelican:** 可能是 Python 中最流行、最为人所知的静态网站生成器。还有很多社区支持, 包括 Pelican 的插件和主题。
- **Hyde:** 稍大于 Pelican。学习曲线要更陡一些, 但是非常健壮。
- **Nikola:** 全特性的、许多社区参与和支持。同时支持自定义, 包括主题和插件。
- **Mynt:** www.pyldadies.com 和一些其他网站在使用。Mynt 认为它在没有硬实现的情况下拥有内容管理系统(CMS)特性。

5.3.2 Web 框架

Web 框架是多合一系统, 它们可以让你创建 API、Web 应用、甚至是综合的 CMS。以下是我们推荐的一些 Web 框架:

- **Flask:** 用途广泛, 从创建简单的其他人可以访问的 API 到创建完整的 Web 应用。
- **Django:** 非常流行也非常健壮。它甚至包含一个管理界面。这个界面允许用户通过高度定制的易用用户界面向数据库添加记录。
- **Bottle:** Bottle 要比 Flask 小, 它只是提供创建网站所需要的功能。它只包含非常少、非常局限的额外功能, 非常适用于更小的网站和网页。
- **Pyramid:** 与 Flask 非常类似, 可以用来创建小型项目。但是也可以用来创建大型项目, 而且也可以根据需求来扩展它。

当然, 还有更多可用的 Web 框架。建议你寻找一个适合你和你的项目的框架, 或者多试试几个框架来寻找一个你觉得最合适的。

5.4 使用 Python 跨网工作

通过网络连接，可以完成许多不同的任务。不管连接是公网、局域网还是私有网络，都可以处理数据、提供网页，甚至远程运行 Python 脚本。本节介绍一些简单的、不同的可以跨网运行 Python 的方式。没有任何示例是可以用于生产环境的，但这应该是你熟悉 Python 可以提供的不同的功能的好办法。

5.4.1 XML-RPC

XML 远程过程调用(XML-Remote Procedure Call, XML-RPC)是一项老技术，但它仍然被用于一些遗留的系统中。之前，这种技术曾大量使用 XML 来处理数据。现在我们使用 JSON 来回传输数据。然而，一些系统仍然使用 XML 并需要远程过程的调用。因为 Python 提供了一个内置的 XML-RPC 模块，下面使用它创建一个服务器，并查看它的工作原理。

在这个“试一试”中，只用几行代码就搭建了一个 XML-RPC 服务器。它会在本地的一个终端上运行并且提供一个简单的 Python 脚本。然后打开另一个 Terminal 窗口并远程运行代码。

试一试：远程运行 Python 代码

这个“试一试”演示了如何通过 XMLRPC 服务器提供本地目录中的文件。

(1) 打开 Python 解释器，从 SimpleXMLRPCServer 模块中导入 xmlrpc.server 对象：

```
Python 3.3.3 (default, Feb 14 2014, 12:35:03)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from xmlrpc.server import SimpleXMLRPCServer
```

(2) 类似于本章之前的 HTTP 代码，需要创建一个服务器对象。还想创建某种系统反馈，它会告诉你代码在正常工作：

```
>>> server = SimpleXMLRPCServer(("localhost", 8080))
```

(3) 用将要运行的代码设置服务器。在这个示例中，传入了一个会产生平方数的简单函数：

```
>>> def square(n):
...     return n * n
...     print("We've got a connection and are listening on port 8080...huzzah!")
```

(4) 接下来，注册函数，这样它可以被即将创建的客户端代码使用：

```
>>> server.register_function(square, "square")
```

(5) 最后，启动服务器：


```
>>> server.serve_forever()
```

现在你应该看到了打印语句输出的端口信息。正在使用 8080 端口进行服务。

不要关闭这个终端或停止进程。需要它继续运行来完成下面的示例部分。

(6) 打开一个新的 Terminal 窗口，启动 Python 解释器，并导入 xmlrpc.client 库：

```
Python 3.3.3 (default, Feb 14 2014, 12:35:03)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import xmlrpc.client
```

如果恰好读过 http.server 库的任何文档，你可能发现还有一个 http.client 库。它允许你通过客户端访问 URL。xmlrpc 库也通过同样的方式进行设置。

(7) 现在想要设置一个服务器代理对象，如下所示：

```
>>> proxy = xmlrpc.client.ServerProxy("http://localhost:8080")
```

(8) 最后，调用远程过程(或者位于另一个 Terminal 窗口中的服务器的 Python 函数)：

```
>>> print("the square root of 3 is %s" % proxy.square(3))
```

应该看到打印出来的字符串 “the square root of 3 is 9”。现在查看 Terminal 窗口，应该看到服务器接收到一个请求并返回状态码 200，如下所示：

```
127.0.0.1 - - [19/Feb/2014 23:14:41] "POST /RPC2 HTTP/1.1" 200 -
```

(9) 如果想要停止服务器，在服务器运行的 Terminal 窗口中，按住 Ctrl+C 即可。

现在，应该停止了服务器并打开了端口。

示例说明

本示例演示了如何设置一个 xmlrpc 服务器和一个 xmlrpc 客户端，并让它们互相通信。这是 Python 在应用中的强大功能。这两个模块都是 Python 内置的模块，可以被立即使用。

许多人会问这些功能在真实世界中有什么用途。XML 处理曾经是我们跨网处理数据的方式，一些组织仍然还在使用这种方法。但我们想要演示的是可以设置 Python 来远程执行程序。所以，如果有一个大的数据文件并且想要远程处理它，可以在远程服务器上设置代码，然后从另一台机器上调用代码。接下来，会看到一些其他可以完成这些任务的示例。

5.4.2 套接字服务器

如果曾经听过任何人讨论 Web 套接字或流数据，就知道他们通常在讨论 TCP 和套接字服务器。这些服务器利用 TCP 或传输控制协议。你可能听说过 TCP/IP，它们其实是一样的东西。当然，Python 有一些内置的库可以帮助你创建 TCP 套接字并使用 Internet 协议在两点之间发送和接收数据。

在这个“试一试”中，会涉及更多的 Python 的内容并使用更加“Pythonic”的方式完

成一些事情。你将再次设置一个服务器和一个客户端，所以需要运行两个 Terminal 窗口来完成这个任务。然而，你也会创建一个类来处理 TCP 请求，以演示 Python 中类的使用。

试一试：通过 TCP 远程执行 Python 代码

这个“试一试”演示了如何通过 Python 设置一个套接字服务器，以便使用 TCP 通过 Internet 直接在一台机器与另一台机器之间发送和接收数据。

(1) 创建一个新文件 `server.py` 并导入 `SocketServer` 模块：

```
#server.py
```

```
import socketserver
```

(2) 为请求处理程序创建类。这个类会在每次连接中被实例化一次。每一次你都想要重写 `handle()` 方法。这样可以与客户端通信：

```
class TCPHandler(socketserver.BaseRequestHandler):
    def handle(self):
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())
```

`self` 是 `TCPHandler` 类的实例。TCP 套接字被连接到客户端上(当看到客户端代码时会更明白)。在此你接收了一个从客户端发送的请求，对它进行 `strip` 操作，并将它保存在实例的 `data` 属性上(`self.data`)。然后调用了一些用于显示的打印语句，这样可以在事情发生时看到它们。最后，将数据发回给客户端，但是把字符串转换为大写字符，这样可以看到发生的改变。

(3) 现在你会做一些熟悉的事情——检查是否在以 `main` 运行。如果是，则设置 `HOST` 和 `PORT` 变量，让代码知道你想为连接监听什么主机地址和端口：

```
if __name__ == "__main__":
    HOST, PORT = "localhost", 8080
```

(4) 最后，你做了些每次创建服务器都会做的一些事情：创建一个服务器对象，让你的客户端连接它，然后永远为它服务：

```
server = SocketServer.TCPServer((HOST, PORT), TCPHandler)
server.serve_forever()
```

现在，需要创建客户端代码。

(5) 创建一个新文件 `client.py` 并导入合适的库：

```
#client.py
```

```
import socket
```

```
import sys
```

(6) 现在需要定义想要连接到的主机和端口——记住，server.py 代码正在监听这个主机和端口上将要到来的连接。

```
HOST, PORT = "localhost", 8080
```

(7) 接下来，创建一些可以发送给服务器的数据。使用 sys.argv 解析你要传入的参数。数据如下：

```
data = " ".join(sys.argv[1:])
```

(8) 现在需要设置套接字。socket.SOCK_STREAM 是将要连接的套接字的类型(TCP)：

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

(9) 现在使用 try:finally 来试着进行连接：

```
try:
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    received = str(sock.recv(1024), "utf-8")
```

```
finally:
    sock.close()
```

在这里，试图连接服务器并发送数据。然后将接收到的数据保存到“received”变量中。最后，不管发生什么(无论获得连接和发送/接收数据失败还是成功)，都关闭了套接字。

(10) 最后，打印连接的结果：

```
print("Sent: {}".format(data))
print("Received: {}".format(received))
```

(11) 运行代码看看会发生什么。再一次，需要两个 Terminal 窗口。在第一个 Terminal 窗口中，运行 Python 代码：

```
$ python server.py
```

(12) 在另一个 Terminal 窗口中，运行客户端代码并传入一个字符串作为参数：

```
$ python client.py Hello from Python Projects' TCP server!
```

(13) 验证以下内容显示在服务器窗口中：

```
$ python server.py
127.0.0.1 wrote:
b'Hello from Python Projects TCP Server!'
```

(14) 验证以下内容显示在客户端窗口中：


```
$ python3 client.py Hello from Python Projects TCP Server!  
Sent:      Hello from Python Projects TCP Server!  
Received:  HELLO FROM PYTHON PROJECTS TCP SERVER!
```

可以连接到服务器，这让服务器很兴奋。

示例说明

本示例成功创建了一个 TCP 套接字并通过一个客户端脚本连接它，然后让每个脚本来回发送和接收数据。通过设置一个服务器并告诉它要监听的主机和端口，可以创建一个客户端。客户端可以将数据发送到主机的那个端口上。将这两部分连接在一起并成功地发送/接收数据。

在此仅演示了 Python 套接字最基本的功能。我们的任务是让你熟悉 Python 生态圈中所有可用的工具。有时候，需要一个直接的数据连接来实时更新数据。这时，套接字和流数据就变得非常方便。幸运的是，Python 让这个事情非常轻松。当然，如果要创建低层网络连接，则建议你在接受任务之前，要先了解需要注意的安全风险和注意事项。

5.5 更多 Python 网络编程的乐趣

你可能想要对 Python 网络编程有更深入的了解。如下是一个可以下载的、较流行的可用网络编程库清单：

- **Twisted**(<http://twistedmatrix.com>): Twisted 非常大也非常强大，并且充满了网络编程的优点。但是在本书成文时，它还不完全支持 Python 3。如果对一些事件驱动的网络编程感兴趣，Twisted 会很适合你。它支持 SMTP、POP3、IMAP、SSHv2 和 DNS。所以，如果想要创建自己的电子邮件服务器，Twisted 将非常适合你。如果对设置自己的 SSL 服务器感兴趣，就使用 Twisted 吧！
- **Tornado**(<http://www.tornadoweb.org/>): Tornado 是一个 Web 框架并且是异步网络库。主要用于需要长连接的大型应用。Tornado 使用非阻塞式 I/O，完美适用于 Websockets 和长轮询。我们将 Tornado 放在网络编程部分而不是 Web 框架部分，是因为它更专注于 Web 框架的网络编程部分而不是创建模板和让东西更漂亮。
- **gevent**(<http://www.gevent.org/>): 根据 gevent 的官方网站，gevent 是一个基于协程的 Python 网络编程库。它提供高级别的同步 API。当想要跨网络做一些疯狂的协程操作时，你可能想要试一试 gevent。当前，gevent 只能用于 Python 2。

现在，你应该对使用 Python 进行跨网络发送和接收数据的能力和便捷有了一个很好的理解。如果对使用 Python 传输数据感兴趣，本章应该已经给了你一个不错的起跳点。你可以探索更多相关内容，并且希望你用自己的发现来创建有趣的内容。

Python 社区中很多人都乐于助人。所以，如果找到了非常喜爱的框架或库，可以加入

邮件列表、IRC 频道，以及任何可以找到的会话。为那些项目做贡献并帮助社区变得更大更强。参与到你感兴趣或对你有帮助的技术中，不但可以帮助提供技术的项目和组织，而且可以帮助所有需要使用或想要学习这项技术的开发者。

5.6 本章小结

本章首先介绍了 Python 如何在 Web 上工作。一个 Web 应用的前端包括处理 HTML、CSS 和 JavaScript 的浏览器。在这里，中间层是 Python。后端包括数据库(SQLite)和 Web 服务器。你也学习了 API，它只是一种允许其他人使用外部应用交互而实际上并不访问数据库本身的方式。接下来，练习使用了一个第三方 API 和 Requests 库。同时，还探索了 Requests 在底层使用的技术。然后，通过使用 FlaskWeb 框架，为借出库添加了一个 Web 界面，并让它可以通过浏览器交互。最后，介绍了一些跨网络运行 Python 的简便方法。

练习

1. 考虑本章前面的这段代码：

```
>>>for result in results['results']:
...     id = result['id']
...     print(id)
...     print (result['id'])
...     details = requests.get(market + id).json()
...     print (details)
...     print (details['marketdetails'])
...     print (details['marketdetails']['GoogleLink'])
```

利用所学的 Python 知识，你能找出一种与上述代码功能相同的列表推导(list comprehension)吗？记住，列表推导的结构如下：

```
[expression for item in list if conditional]
```

2. 利用在 Python 中使用文件的知识，你能将 USDA 的 API 的输出保存到机器上的一个文件中以供后续处理吗？(将它存为 .txt 文件合适吗？)

3. 你能在 Flask 文档中找到关于把应用拆解为更小的模块化文件的信息吗？这样端点或视图就在单独的文件中而不是让所有东西都在一个大的 Python 文件中(提示：这是 Flask 提供的一个概念/特性)。

4. 你能找到其他的 HTTP 方法吗？你能找到在 Flask 应用中使用它们的方法吗？

5. 通过阅读 Requests 文档，你能找到通过给请求方法传入 URL 就能将一个网站输出为 HTML 文件的方法调用吗？

本章所学知识

主 题	关 键 概 念
Python 在 Web 上的应用	服务器使用 Python 语言来操纵 Web 上多台机器之间相互传输的实际数据
HTTP 和 HTTP 方法	机器在 Web 间通信的基本语言。GET 和 POST 方法及其用途
API	应用程序编程接口，或与其他服务器交互来获取或操纵数据的方式。可以创建或使用 API
如何创建 Flask 应用	Flask 框架功能非常强大，它使用 Python 作为服务器端代码来创建 Web 应用。这也演示了如何创建 API
Flask 中的模板	模板在 Web 应用中非常常用。它们可以在网页上引入逻辑并创建动态内容
XMLRPC	Python 通过内置的功能来创建 XMLRPC 服务器，以实现在 Web 上传输数据。这只对那些支持旧的或遗留系统的开发者有帮助。这些开发者仍然在使用这种方法传输数据
套接字服务器	Python 有一个非常好用的内置库，即 socketserver。它允许你通过其他脚本创建套接字来进行连接。这对于跨网络处理数据是非常强大的
SimpleHTTPServer	这是一个创建简单 HTTP 服务器的方法。它可以用于本地测试文件或被集成到更大的框架中作为调试模式来使用

第 6 章

Python 在更大项目中的应用

本章主要内容:

- 测试 Python 代码
- 调试 Python 代码
- 处理 Python 代码中的错误
- 组织并发布 Python 代码
- 调试 Python 代码的性能

从 wrox.com 下载本章代码

可以在 wrox.com 网站找到本章涉及的代码。代码可以在 www.wrox.com/go/python-projects 的 Download Code 选项卡下找到。第 6 章代码位于 Chapter 6 download, 名为 Chapter6.zip, 每个文件都是根据本章提到的代码文件名命名的。

迄今为止, 你已经了解了很多种使用 Python 的方法。已经创建了小的脚本来处理小型任务, 已经在本地处理了中型任务, 甚至已经使用 Flask 创建了一个小型 Web 应用。但是如果你发现自己处于一个更大项目中时, 该怎么办呢? 至此你已经看到, Python 是一门非常强大的语言。它也非常开放, 这意味作为开发者的你可以使用语言的所有方面。然而, 这种开放也让测试你的 Python 代码变得比以往更加重要。Python 中的每个对象都是一级对象, 所以可以更改和操纵任何可用的对象。因为可以改变和操纵对象, 所以必须确保测试和验证我们代码的逻辑。

Python 不是强类型的语言, 不需要像 C 和 Java 那样显式地指定类型。在 Python 中可以传递对象, 而解释器会尽力试图操纵它们。然而, 如果它不能对可用对象或数据执行一个操作, 就会抛出一个异常, 这会让你的程序崩溃。所以, 如何能阻止它发生呢? 如何编写代码, 分享代码, 并保证其他人可以使用它, 而代码会像期待的那样作用呢? 测试。

6.1 使用 doctest 模块测试

在 Python 中，最简单的测试形式是 doctest 模块。这个模块用于测试代码的稍简单部分，以验证它们像期待的那样工作。把测试代码写在文档字符串(三引号"..."或"..."，单或双引号都可以)中。doctest 的写法如下：

```
'''
this function should take in a number and return its squared value
>>> sq(3)
9
'''

def sq(n):
    return n*n
```



注意：doctest 会要求代码缩进。第一行代码的缩进决定后续的缩进，所以需要遵循这个缩进模式。doctest 字符串会像编写的那样精确地传入解释器——如果解释器期待一个特定的缩进，则需要确保 doctest 字符串是符合那个缩进模式的。

同时要记住的是，随着 Python 的改变和进化，缩进模式可能会改变，所以 doctest 字符串可能在将来不可用。这就是为什么许多人在很重要的测试中不很依赖于 doctest 的原因之一。

编写 doctest 测试最常用的方式是使用解释器，编写代码，然后在解释器中运行它。之后将解释器中的文本复制并粘贴到 doctest 字符串中，如下所示：

```
Python 3.3.3 (default, Feb 14 2014, 12:35:03)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def sq(n):
...     return n*n
...
>>> sq(3)
9
>>>
```

所以，应该简单地复制下面几行，然后把它们放入 doctest 字符串中：

```
>>> sq(3)
9
```

doctest 不适合测试大的、复杂的方法或函数。但是，它非常擅长“约定编程(contract

programming)”。通过使用 doctest 字符串并说：“这个函数，当传入参数 3 时，会返回 9”，然后调用函数，就设置了一个约定：“如果传递一个特定的数据片，函数会像期待的那样执行”。然而，不能测试所有可能的输出，所以 doctest 在较大项目中很快就会遇到瓶颈问题。

在接下来的示例中，会创建并执行一个小的 Python 脚本，并使用 doctest 字符串来测试代码。

试一试：创建并执行简单 doctest 测试

这个“试一试”演示了如何使用 doctest 测试一个有几个函数的简单文件。这个 doctest 测试使用三引号("""...)把测试字符串放在文档字符串中。

(1) 在项目目录中为第 6 章创建一个目录，然后使用自选的编辑器创建一个名为 simple_doctest.py 的 Python 文件。该文件包含如下函数和测试：

```
def simple_math(x, y):
    """
    >>> simple_math(1, 2)
    3

    >>> simple_math('k', 'v')
    'kv'
    """

    return x + y
```

为了进行测试，必须在解释器提示符(>>>)后有一个空格。包含解释器提示符(>>>simple_math(1,2))的第一行被格式化为>>>simple_math(1,2)，它将不能正确运行。空格是强制的。

(2) 从第 6 章的目录中打开一个 Terminal 窗口，执行如下命令：

```
python -m doctest -v simple_doctest.py
```

这里，在调用 Python，但是通过传递一个-m 标签，告诉 Python 你想要使用一个模块——在此是 doctest 模块——来执行文件。而-v 标签意味着你想要完整的输出。如果去掉-v 标签并返回代码，你会看到它只是悄悄地结束，这意味着代码运行了，但是你获得的是另一个 Terminal 提示符而 Python 解释器没有返回其他的东西。最后一个参数当然是指你正在测试的文件。通过-v 标签，应该看到如下输出：

```
~chapter6$ python -m doctest -v simple_doctest.py
Trying:
    simple_math(1, 2)
Expecting:
    3

ok
Trying:
```



```

    simple_math('k', 'v')
Expecting:
    'kv'
ok
1 items had no tests:
    simple_doctest
1 items passed all tests:
    2 tests in simple_doctest.simple_math
2 tests in 2 items.
2 passed and 0 failed.
Test passed.

```

注意，为了让测试运行，必须在解释器提示符(>>>)后加上一个空格。

(3) 接下来，编写这些测试，就像你想要完全文档化的约定编程。在编辑器中，打开 `simple_doctest.py` 文件并添加如下行：

```

def simple_math(x, y):
    '''
    This function will return x + y
    we can use it on numbers. Passing 1 and 2:

    >>> simple_math(1, 2)
    3

    We should get 3 as a return value

    It will also work on strings. Passing the strings 'k' and 'v':

    >>> simple_math('k', 'v')
    'kv'

    We should get 'kv'
    '''
    return x + y

```

注意，必须在期待的结果和放入 `doctest` 字符串中的任何文档字符串之间有一个换行。所以，当调用 `simple_math(1,2)` 后会期待获得 3，则必须在指定你想要的行为之前换行。否则，解释器会试图把那一行作为期待的输出，因此认为测试失败。

(4) 有时，需要测试一个不能一直保持一致的预测值(就像一个内存地址)。在 `simple_doctest.py` 文件中加入如下代码(在第一个测试通过之后)：

```

class SimpleClass():
    pass

def class_testing_method_ahoy(obj):
    ''' Should return a list containing the object

    >>> SimpleClass(class_testing_method_ahoy())
    [<doctest_class_testing_method_ahoy.SimpleClass object at /

```

```
0x10382a390]
```

```
'''
```

```
return [obj]
```

现在执行测试并观察输出。你应该看到测试失败了，因为代码在检查一个我们不能每次都准确预测的内存地址。请注意输出中的内存地址。

```
chapter6 $ python -m doctest -v simple_doctest.py
Trying:
    class_testing_method_ahoy(SimpleClass())
Expecting:
    [<doctest_class_testing_method_ahoy.SimpleClass object at /
    0x10382a390>]
*****
File "./simple_doctest.py", line 27, in /
simple_doctest.class_testing_method_ahoy
Failed example:
    class_testing_method_ahoy(SimpleClass())
Expected:
    [<simple_doctest.SimpleClass object at 0x10382a390>]
Got:
    [<simple_doctest.SimpleClass object at 0x10af0fe50>]
Trying:
    simple_math(1, 2)
Expecting:
    3
ok
Trying:
    simple_math('k', 'v')
Expecting:
    'kv'
ok
2 items had no tests:
    simple_doctest
    simple_doctest.SimpleClass
1 items passed all tests:
    2 tests in simple_doctest.simple_math
*****
1 items had failures:
    1 of 1 in simple_doctest.class_testing_method_ahoy
3 tests in 4 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
```

doctest 需要实际输出精确地匹配所期待的输出。当在我们为 doctest 的期待输出指定为一个内存地址时，测试中取得的实际内存地址必须与期待的值完全一致。当我们：

```
[<simple_doctest.SimpleClass object at 0x10382a390>]
```

`doctest` 想要内存地址为 `0x10382a90` 上的对象，但是你会在一个新的内存地址上创建一个新对象。其实，你并不在乎内存地址，你只在乎被创建的对象。为此，`doctest` 提供了一个方法：

```
>>> class_testing_method_ahoy(SimpleClass()) /
# doctest: +ELLIPSIS
[<simple_doctest.SimpleClass object at 0x...>]
```

`ELLIPSIS` 选项让 `doctest` 知道之后的内容可以是任意值。这会返回一个成功的测试：

```
Trying:
    class_testing_method_ahoy(SimpleClass()) # doctest: +ELLIPSIS
Expecting:
    [<simple_doctest.SimpleClass object at 0x...>]
ok

Trying:
    simple_math(1, 2)
Expecting:
    3
ok

Trying:
    simple_math('k', 'v')
Expecting:
    'kv'
ok

2 items had no tests:
    simple_doctest
    simple_doctest.SimpleClass
2 items passed all tests:
    1 tests in simple_doctest.class_testing_method_ahoy
    2 tests in simple_doctest.simple_math
3 tests in 4 items.
3 passed and 0 failed.
Test passed.
```

如果正在检查是否返回了一个列表，`ELLIPSIS` 常量也同样有用，比如当使用 `range()` 方法时。比如说，你想要确保当调用 `range(4589)` 时，得到了数字 1-4590。可以使用 `ELLIPSIS` 常量并只是设置你的结果为 `[0, 1, ..., 4588, 4589]`，而不是打印整个 4590 个数字的整个列表。`doctest` 包含许多这种适用于不同情况的常量。可以在完整的 `doctest` 文档中查看所有的常量。

示例说明

`doctest` 模块是 Python 语言内置的模块。它接受通常直接从解释器中复制出来的字符串，然后在文件被调用时检查那些字符串。它通过使用模块来实现这一点(在命令行调用 Python，其方法是使用 `-m` 标签，接着是模块 '`doctest`'，最后是文件名)。

尽管 `doctest` 非常适合检查你的文档字符串是否为真，以及代码的行为是否如期待的那样，但是它不适用于对复杂的代码库进行彻底的健壮性测试。`doctestAPI` 还有很多其他的内容。应该查看文档来熟悉该模块的全部功能。

6.2 使用 unittest 模块测试

如果需要重要的测试而且想要验证代码如你期待的那样运行，该怎么办呢？这就是 `unittest` 模块的工作。这个模块要比 `doctest` 模块更加健壮，而且会彻底地测试代码。`unittest` 就像底线测试模块，大多数测试库都基于它。在 Python 中，它对于测试驱动开发(test-driven development, TDD)也是一个优秀的介绍。

术语 `unit test` 并不是 Python 特有的。如果熟悉其他语言和编程，毫无疑问应该听说过单元测试。单元测试只是按单元测试代码。所以，如果在代码中有 5 个函数，为了测试代码库中每个单元的功能，则要在测试组中有至少 5 个单元。单元测试也包含一个测试文件，其中包含了所有的测试，该测试文件的结构或格式与其他任何 Python 文件的相同。唯一区别在于每个测试以 `test` 开头，并且每个测试组是 `unittest.TestCase` 对象的一个类。例如，如果你有一个名为 `login` 的函数，并且想要测试这个函数，那么创建一个名为 `test_login` 的测试，之后它会调用 `login` 函数并对该函数的输出进行测试。

在编写 `unittest` 类时，不要忘了需要在测试代码中导入要测试的模块。如果正在测试 `users.py`，则需要将 `import users` 添加到 `test.py` 文件中。这样可以通过单元测试来测试 `users` 模块中的函数。

通过创建 `TestCase` 类的子类，创建了 `unittest` 测试，如下所示：

```
import unittest

class PythonProjectsTest(unittest.TestCase):
    return
```

你想要把语句放入类中。当测试运行时，会检查这个类并返回一个断言值 `True` 或 `False`：

```
import unittest

class PythonProjectsTest(unittest.TestCase):
    def test_to_fail(self):
        self.failIf(False)

if __name__ == '__main__':
    unittest.main()
```

在之前的示例中，使用了断言方法 `failIf()` 来检查括号中的值。如果值为真，当运行测试时，将会收到一个错误信息。在这个示例中，传入了 `False`，这当然会被认为是假。因此，这个测试会返回一个失败。

如果运行这个测试，应该看到如下输出：

```
=====
FAIL: test_to_fail (__main__.PythonProjectsTest)
-----
```

```
Traceback (most recent call last):
  File "<stdin>", line 3, in test_to_fail
AssertionError: True is not false
```

```
-----
Ran 1 test in 0.000s
```

```
FAILED (failures=1)
```

如果将 `self.failIf(True)` 修改为 `self.failIf(False)`，应该看到输出修改为：

```
-----
Ran 1 test in 0.000s
OK
```

注意，`unittest` 并不检查一个测试是否被真的执行了。它仅检查是否有异常被抛出。因此，如果没有异常被抛出，测试被认为 **OK**。这可能意味着你的精确计算在返回的并不是十分精确的值时仍然显示通过或者 **OK**。这并不是因为结果是正确的，其实并不正确，只是因为测试没有抛出异常而已。

如果真没有通过测试，如下是三个可能的单元测试的输出：

- **OK**：测试是 **OK** 的，没有抛出异常。
- **Fail**：抛出 `AssertionError`(测试失败)。
- **Error**：抛出非 `AssertionError` 异常。

理解单元测试和 `unittest` 模块的最好方式就是做一些测试。

试一试：使用 `unittest` 模块创建和运行单元测试

在这个“试一试”中，你会编写函数并使用 `unittest` 模块测试它们，以便理解 `unittest` 模块的架构。

(1) 在 Chapter 6 目录中，创建 `ch6_example.py` 文件。这个文件包含一些其实没有用的函数，但是它们非常易于测试：

```
#ch6_example.py

def first(chars):
    chars.sort()
    return chars[0]

def last(chars):
    chars.sort()
    return chars[-1]
```

(2) 创建名为 `unittest_example.py` 的测试文件。导入 `unittest` 模块，然后从 `ch6_example.py` 文件导入两个函数。直接导入这些函数意味着在测试时不需要调用 `ch6_example.first()` 或 `ch6_example.last()`。可以仅调用 `first()` 和 `last()`。记住，这被称为通过导入为函数在代码中设置别名。

```
#unittest_example.py

import unittest
from ch6_example import first, last
```

(3) 创建两个列表，一个是数字列表，另一个是字符串列表。将会使用这些列表测试两个不同的排序函数。然后，创建从 `unittest.TestCase` 类继承的测试类：

```
#unittest_example.py

import unittest
from ch6_example import first, last

list_nums = [7,9,5]
list_chars = ['m', 'd', 'Z', 'l']

import unittest

class TestPPMath(unittest.TestCase):
```

(4) 接下来，测试一些断言来看看它们是如何表现的。我们从最常用的断言 `assertEqual` 开始。这个测试应该会通过，因为当你对数字列表排序时，列表中的第一个元素是 5，所以这应该会返回真：

```
import unittest
from ch6_example import first, last

list_nums = [7,9,5]
list_chars = ['m', 'd', 'Z', 'l']

class TestPPMath(unittest.TestCase):

    def test_first(self):
        self.assertEqual(first(list_nums), 5)
```

记住：想要运行的所有测试函数必须以 `test` 开头。

(5) 类似于检查相等性的 `AssertEqual`，还有一个 `assertTrue`。它会检查第一个值是否是第二个值，如果是，则为真：

```
import unittest
from ch6_example import first, last

list_nums = [7,9,5]
list_chars = ['m', 'd', 'Z', 'l']
```



```
class TestPPMath(unittest.TestCase):

    def test_first(self):
        self.assertEqual(first(list_nums), 5)

    def test_last(self):
        self.assertTrue(last(list_chars), 'm')
```

(6) unittest 只会寻找异常，比如 `assertionError` 异常。可以使用 `failUnless()` 函数来告诉它，除非它返回真，否则测试失败：

```
import unittest
from ch6_example import first, last

list_nums = [7,9,5]
list_chars = ['m', 'd', 'Z', 'l']

class TestPPMath(unittest.TestCase):

    def test_first(self):
        self.assertEqual(first(list_nums), 5)

    def test_last(self):
        self.assertTrue(last(list_chars), 'm')

    def testFirstAgain(self):
        self.failUnless(first(list_chars), 'Z')
```

(7) 如果想要在它是真时让测试失败，可以使用 `failIf()` 函数。它在输入被认为是真时失败。所以，这个测试在运行时应该会失败：

```
import unittest
from ch6_example import first, last

list_nums = [7,9,5]
list_chars = ['m', 'd', 'Z', 'l']

class TestPPMath(unittest.TestCase):

    def test_first(self):
        self.assertEqual(first(list_nums), 5)

    def test_last(self):
        self.assertTrue(last(list_chars), 'm')

    def testFirstAgain(self):
        self.failUnless(first(list_chars), 'Z')

    def testLastAgain(self):
        self.failIf(last(list_nums), 9)
```

(8) 最后, 插入 `__main__`, 检查并运行 `unittest.main()` 方法来真正测试新测试类:

```
import unittest
from ch6_example import first, last

list_nums = [7,9,5]
list_chars = ['m', 'd', 'Z', 'l']

class TestPPMath(unittest.TestCase):

    def test_first(self):
        self.assertEqual(first(list_nums), 5)

    def test_last(self):
        self.assertTrue(last(list_chars), 'm')

    def testFirstAgain(self):
        self.failUnless(first(list_chars), 'Z')

    def testLastAgain(self):
        self.failIf(last(list_nums), 9)
if __name__ == '__main__':
    unittest.main()
```

示例说明

当编写测试时, 只是创建了一些静态数据, 这些数据会被传入到已经定义的函数中。你想要向函数传入一个已知的值, 然后在测试中展示期待得到的返回值。如果那个值没有被返回, 测试应该会失败。如果值返回了, 测试会通过, 并且代码会转向下一个测试函数。

一些读者可能很快意识到测试静态数据并不是不会出错。如果传入的数据不是你测试的类型怎么办? 这就是为什么编写好的测试非常重要的原因所在。程序中的一个函数可能有多个测试, 或者一个测试可以验证多种情况。

6.3 Python 中的测试驱动开发

在 Python 社区中, 一个越来越流行的术语是测试驱动开发(test-driven development, TDD)。它到底是什么意思呢? 尽管 TDD 在 Python 开发中是一个非常重要的话题, 但它也是一个非常健壮的话题。因此, 这部分只会对 TDD 进行一个简要的介绍, 这样可以熟悉一下这个术语以及它的基本定义。

TDD 只是意味着要首先编写你的测试。大多数开发者在听到测试这个词时会有所怨言。他们认为这意味着更长的开发时间以及更多的努力, 以及更少有趣的东西, 比如编写让项目运行的实际代码。然而, 测试可以和其他东西一样有意思。并且尽管它要求开发者写更多行的代码, 但是它也会产生更高质量的代码并且之后项目具有更好的可维护性。在将来, 你的同事都会感谢你花费时间先写测试并根据这些测试进行开发。

所以，TDD 到底是如何工作的呢？编写测试！它真的就这么简单。当然，也有编写良好测试的艺术形式，你应该花一些时间学习和熟悉适当的 TDD 实践，这非常重要。以下是一些基本常识：

- (1) 首先编写测试。
- (2) 一开始所有测试都应该失败。
- (3) 编写代码。
- (4) 用测试检验代码。
- (5) 重写代码。
- (6) 再次用测试检验代码。
- (7) 重复所有测试直到通过。

这是 TDD 的主旨。你可能明白了为什么 `doctest` 可能并不适用于所有测试情况。一旦你必须测试和再次测试以及开始测试更加复杂的方案，`doctest` 就会遇到瓶颈。如上所述，没有编写有效测试的艺术，然而这也是 TDD 的美妙之处。

6.4 调试 Python 代码

大多数开发者可能会告诉你他们讨厌调试。调试是乏味的、挑剔的，而且可以很快就变得非常无聊或令人恼怒。其实，它并不是必须如此。重新审视一下调试和测试可以让甚至最愤青的开发者稍微息怒一些。

当开始遇到代码中的 `bug` 时，不要想它是多么烦人(不要担心，这是自然反应)，要认为这实际上是一个学习的机会。代码中的某处出现了问题，那肯定是有些地方你没有研究透彻。这是你透彻学习那个地方的机会，找到 `bug` 的原因，看看到底是因为什么！伴随着你处理的每个 `bug`，你正走在成为经验丰富的程序员的道路上。

Python 通过 Python 调试模块或 `pdb` 让调试变得简单。如果你读过第 5 章并探索过 Chrome 开发者工具，可能注意到一些相同之处。如果你的职业是一个 Web 开发者并且试图试一试 Python，可能会发现你喜欢 `pdb`，而且它会让你想起一些你最喜欢的网络调试软件。

`pdb` 相当强大。它可以帮助你代码中插入断点。断点可以停止代码的运行，并让你进入一个 `pdb` 提示符或终端。这是非常方便的，因为你可以开始检查在那一刻的范围内你拥有的数据。如果发现当一个特定函数被调用时有一个异常抛出，则可以在那个函数内放置一个 `pdb()` 调用。之后可以在终端的交互式解释器中检查数据。让我们试一试吧。

下面的示例展示了如何使用 `pdb` 模块调试 Python 代码。

试一试：使用 Python 调试器或 `pdb` 模块(`pdb_example.py`)

这个“试一试”展示了如何使用 `pdb` 模块的能力来调试或检查 Python 代码。

- (1) 打开 `pdb_example.py` 文件。你应该看到如下内容：

```
#pdb_example.py
```



```

class ExampleClass(object):

    def __init__(self, name, number):
        self.name = name
        self.number = number

    def example_entry(self):
        return "The example name is {0} with the number {1}".format(self.name,
            self.number)

if __name__ == '__main__':
    example = ExampleClass("Carla", 456)
    return example.example_entry()

```

(2) 导入 pdb 模块:

```

#pdb_example.py

import pdb

class ExampleClass(object):

    def __init__(self, name, number):
        self.name = name
        self.number = number

    def example_entry(self):
        return "The example name is {0} with the number {1}".format(self.name,
            self.number)

if __name__ == '__main__':
    example = ExampleClass("Carla", 456)

    return example.example_entry()

```

(3) pdb 模块有许多强大的功能。看到的第一个方法是 `set_trace()` 方法，所以将 `set_trace()` 方法添加到代码中：

```

#pdb_example.py

import pdb

class ExampleClass(object):

    def __init__(self, name, number):
        self.name = name
        self.number = number

    def example_entry(self):
        pdb.set_trace()

```

```

        return "The example name is {0} with the number {1}".format(self.name,
self.number)

if __name__ == '__main__':
    example = ExampleClass("Carla", 456)

    return example.example_entry()

```

(4) 保存文件。现在运行 `pdb_example.py` 文件。应该进入到 `pdb` 解释器中。它的提示符为(`pdb`)提示符:

```

chapter6$ python pdb_example.py
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py(13)
example_entry()
-> return "The example name is {0} with the number {1}".format( self.name,
self.number)
(Pdb)

```

(5) 输入 `n` 并按 `Enter/Return` 键:

```

(Pdb) n
--Return--
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py(13)
example_entry()->'The example ...he number 456'
-> return "The example name is {0} with the number {1}".format( self.name,
self.number)
(Pdb)

```

你刚才的动作是在文件中向下走一行(`n`)。查看 `pdb_example.py` 文件, `set_trace()`被放置在返回字符串之前:

```

#pdb_example.py

import pdb

class ExampleClass(object):
    def __init__(self, name, number):
        self.name = name
        self.number = number

    def example_entry(self):
        pdb.set_trace()
        return "The example name is %s with the number %d" % name, number

if __name__ == '__main__':
    example = ExampleClass("Carla", 456)

    example.example_entry()

```

这意味着程序会在那行中断并启动 `pdb` 解释器, 这样就可以检查代码。当输入 `n` 然后

按 Enter/Return 键时，你会移动到代码的下一行，也就是返回语句那一行。该行语句会执行，且会看到字符串被打印出来(添加了一些省略号来表示为了可读性而省略的文本)：

```
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py (13)
example_entry() > 'The example ...he number 456'
-> return "The example name is {0} with the number {1}".format( self.name,
self.number)
(Pdb)
```

(6) 当仍然在调试器中时，再按一下 Enter/Return 键，你应该看到代码中的下一行语句被执行。这样做的效果就像你输入 `n` 并按 Enter/Return 键一样。调试器会保存上一个命令并且会在按 Enter/Return 键时就执行它。

```
(Pdb)
--Return--
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py (19)
<module>()
->None
-> example.example_entry()
(Pdb)
```

如果继续按 Enter/Return 键，会看到余下的程序一步步执行直到它完成。你会回到 Python 命令提示符中并退出 pdb 环境：

```
(Pdb)
--Return--
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py (19)
<module>()
->None
-> example.example_entry()
(Pdb)
chapter6$
```

(7) 再一次启动调试器，试试一些更方便的命令：

chapter6\$ python pdb_example.py

```
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py (13)
example_entry()
-> return "The example name is {0} with the number {1}".format( self.name,
self.number)
(Pdb)
```

(8) 这一次，打印一些变量值。在调试器提示符上，输入 `p self.name` 并按 Enter/Return 键：

```
> /Users/lcassell/Documents/Python_Companion/chapter6/pdb_example.py (13)
example_entry()
-> return "The example name is {0} with the number {1}".format( self.name,
self.number)
(Pdb) p self.name
'Carla'
```


(Pdb)

可以通过输入 `p` 后跟变量名来使用打印功能。

(9) 在提示符上, 输入 `locals()` 并按 `Enter/Return` 键。应该看到在此时局部作用域内的所有对象:

```
(Pdb) p self.name
'Carla'
(Pdb) locals()
{'self': <__main__.ExampleClass object at 0x106c66450>}
(Pdb)
```

在本示例中, 当前本地作用域内只包括类, 这是理所应当的。

(10) 输入 `globals()` 并查看全局作用域内有哪些可用对象:

```
(Pdb) locals()
{'self': <__main__.ExampleClass object at 0x106c66450>}
(Pdb) globals()
{'example': <__main__.ExampleClass object at 0x106c66450>, '__builtins__':
<module
'builtins' (built-in)>, '__name__': '__main__', '__file__':
'pdb_example.py',
'ExampleClass': <class '__main__.ExampleClass'>, 'pdb': <module 'pdb' from
'/usr/local/Cellar/python3/3.3.3/Frameworks/Python.framework/Versions/3.3/
lib/python3.3/pdb.py'>, '__package__': None, '__loader__':
<_frozen_importlib.
SourceFileLoader object at 0x106b9a410>, '__cached__': None, '__doc__': None}
(Pdb)
```

注意, 你有很多可用的对象, 包括 `ExampleClass` 对象、`pdb` 模块(你导入的)和本地 Python 源。有时在调试时, 需要查看本地作用域内有什么, 以此检查是否有可用的数据。在这些情况下, `locals()` 和 `globals()` 将会非常有用。

(11) 输入 `c` 并按 `Enter/Return` 键。这时应该退出了 `pdb`, 代码应该结束了, 而应该看到正常的命令提示符。在 `pdb` 中, `c` 命令只是继续运行程序。

(12) 如果不想执行余下的程序就退出调试器, 在 `(pdb)` 提示符中输入 `q` 并按 `Enter/Return` 键。

示例说明

`pdb` 是 Python 标准库中内置的模块。只需要在文件中导入 `pdb`, 然后既可以调用 `stack_trace()` 方法进入调试器环境, 也可以调用其他方法在运行时执行文件内的一些特定函数。这会让你进入调试器界面。`pdb` 在运行时调试代码以及在实际环境中的特定点上检验代码中的数据是非常有用的。`pdb` 模块包含许多命令。你可以在文档中查看更完整的命令列表。

在 Python 中处理异常

Python 是一门解释性语言，这意味着没有编译器去编译代码，并且能够在运行它之前发现任何逻辑或语法错误。那么，Python 是如何处理这种情况的呢？Python 使用异常来处理错误。这种处理类型意味着代码中的一个小错误可能引起整个程序崩溃。因此，你可能希望进行彻底的测试。但是在此之上，你也希望设置一些自动防故障的措施，以防代码在运行时碰到异常。

例如，如果在解释器中试试如下代码：

```
>>> def sum(a, b):
...     return a + b
...
>>> sum("no", 4)
```

将会得到如下错误：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
TypeError: Can't convert 'int' object to str implicitly
```

如你所见，当试图向一个数学函数传递字符串，这个函数只能处理整数或浮点数时，会抛出一个 `TypeError` 异常。这告诉你向函数发送的数据类型不正确。由于 Python 不是一门强类型语言并且它也不被编译，因此将会获得的唯一错误就是这些异常会在运行时发生。当异常在运行时被抛出时，如果该异常没有被及时处理，整个程序就会退出。迫切的是，你要检查这种类型的错误。不检查这些错误会导致代码不可用，而这些代码也不能作为非常好的代码库版本！

Python 语言内置了多种异常。以下是这些异常的一个列表：

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StandardError
        +-- BufferError
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
```

```

+-- EnvironmentError
|   +-- IOError
|   +-- OSError
|       +-- WindowsError (Windows)
|       +-- VMSError (VMS)
+-- EOFError
+-- ImportError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- MemoryError
+-- NameError
|   +-- UnboundLocalError
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
+-- SyntaxError
|   +-- IndentationError
|   +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
|   +-- DeprecationWarning
|   +-- PendingDeprecationWarning
|   +-- RuntimeWarning
|   +-- SyntaxWarning
|   +-- UserWarning
|   +-- FutureWarning
|   +-- ImportError
|   +-- UnicodeWarning
|   +-- BytesWarning

```

很多地方都会出错，那么该如何优雅地在 Python 中处理异常呢？使用 try-except 块。try-except 块会试图执行一块代码，如果代码抛出上述异常中的一个，它会捕捉那个异常并打印一条错误消息。这条消息被定义在异常基类中，或者甚至可以为每个异常打印自己的错误消息：

```

>>> try:
...     sum("yes", 9)
... except TypeError:
...     print("Both inputs must be integers")
...
Both inputs must be integers

```

也可以使用 try-except 块处理异常，这样程序就不会崩溃并且可以继续执行程序：

```

>>> try:
...     some_function()
... except:

```



```
...     graceful_function()
... else:
...     next_function()
```

有时无论 try-catch 是否捕捉到一个异常，都希望运行一个函数。在这种情况下，想要使用 finally 语句。

```
>>> try:
...     some_function()
... except:
...     graceful_function()
... finally:
...     cleanup_function()
```

但是，如果想要让代码抛出它自己的异常，该怎么办呢？如果想要检查数据的特定类型，但并没有现有的异常类型而你却想要警告用户，该怎么办呢？可以基于内置创建自定义异常类。

在接下来的示例中，将创建和使用自定义异常。

试一试：在 Python 中创建和使用自定义异常(exceptClass.py)

这个“试一试”展示了如何在 Python 代码中创建并使用自定义异常。

(1) 打开 exceptClass.py，熟悉你将会使用的类：

```
# exceptClass.py

class TestClass(object):

    def __init__(self, name, number):

        name = self.name
        number = self.number

    def return_values(self):

        print ("The values are: ", self.name, self.number)
```

(2) 编写在 self.number 不是一个数字时会抛出的异常。编写异常的第一步是该异常必须是一个继承于 Exception 类的子类。向 exceptClass.py 文件中添加如下行：

```
# exceptClass.py

class TestClass(object):

    def __init__(self, name, number):

        self.name = name
        self.number = number

    def return_values(self):
```

```

print ("The values are: ", self.name, self.number)

class notANumber(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)

```

这里，已经创建了自定义异常。如果 `number` 属性实际上不是一个数字，这个异常就会被抛出。也为 `Exception` 类重写了 `__init__` 函数，并且使用 `value` 而不是 `args` 捕捉了异常中抛出的值。也重写了 `__str__()` 方法，调用 `repr()` 方法输出了 `self.value` 属性。对于引发异常的值，`repr()` 方法给出该值的正确的表示形式(这也就是指打印出来的异常错误消息)。

(3) 接下来，修改 `return_values()` 方法，让它可以检查 `self.number` 是否是 `int` 类型。如果 `self.number` 的类型不是 `int`，则抛出异常。可以在 `try/catch` 中实现一个非常简单的 `if/else` 语句来进行这个检查：

```

# exceptClass.py

class TestClass(object):

    def __init__(self, name, number):

        self.name = name
        self.number = number

    def return_values(self):
        try:
            if (type(self.name) is int):
                return "The values are: ", type(self.name), type(self.number)
            else:
                raise notANumber(self.number)
        except notANumber as e:
            print("The value for number must be an int you passed: ", e.value)

class notANumber(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)

```

在此仅检查了 `self.name` 的类型。如果它不是 `int` 类型，就会抛出之前定义的异常。如果是，则返回一个字符串，告诉你实例的每个属性的类型。如果类型为 `int`，就会抛出 `notANumber`。你会向异常传入被检查的 `self.number` 并打印错误消息。

(4) 现在在交互式模式下运行脚本。启动 Python 解释器，但是使用 `-i` 标签来执行它并

调用 `exceptClass.py` 文件，如下所示：

```
$ python -i exceptClass.py
>>>
```

当使用 `-i` 标签启动 Python 解释器时，可以传入一个 Python 文件。而这会导入你传入的文件，而不必显式地在解释器中导入。这意味着你拥有了在 `exampleClass.py` 文件中定义的两个类，并且不必使用 `exampleClass.<foo>` 调用它们。你可以直接调用它们。

(5) 接下来，创建一个新 `TestClass` 实例，并传入两个字符串(而不是一个字符串和一个整数)：

```
(ch3Ex2)$ python -i exceptClass.py
>>> exampl = TestClass('string1', 'string2')
```

(6) 在新创建的实例中调用 `return_values()` 并注意输出：

```
(ch3Ex2)$ python -i exceptClass.py
>>> exampl = TestClass('string1', 'string2')
>>> exampl.return_values()
The value for number must be an int you passed: string
```

`try-except` 在此发挥了作用，捕捉到你传入的是一个字符串而不是一个整数。

(7) 创建另一个实例并传入一个字符串和一个整数，然后在这个实例上调用 `return_values()` 并注意输出：

```
>>> exm = TestClass('string1', 42)
>>> exm.return_values()
('The values are: ', <class 'str'>, <class 'int'>)
```

示例说明

当创建一个异常类时，实际上创建了 Python 内置的基类 `Exception` 的子类。由此，可以控制自定义异常在被抛出时的表现。创建了一个非常简单的类，并且看到了当异常被抛出时，你的类会向用户反馈传入类中的数据的类型。

如你所见，这个特性在编写大项目时可以非常强大。希望本节的讲解已经让你对异常的形成有了足够的了解。当需要时，可以编写自己的异常。

6.5 工作在更大的 Python 项目中

在使用 Python 进行开发时，你可能会发现不同的项目有不同版本的不同包。当本地环境是 Python 2.7，但你想要工作的项目是 2.6 或 3.4 时，该怎么办呢？这是许多 Python 开发者曾经遇到过的问题，所以他们设计了一个解决方案：使用 `virtualenv` 环境。

`virtualenv` 是一个 Python 项目的虚拟环境。它有助于你创建多个 Python 实例并为特定

项目开发需要的所有库。假如你想要在一个使用 Python 2.7 的项目上工作，已经在本地安装了它，但是项目需要一个与你本地安装的版本不同的库。Python 版本吻合但是库的版本不吻合，该怎么办呢？这就是 `virtualenv` 要做的工作！

在这个示例中，你会创建并激活 `virtualenv` 来为单独的 Python 项目创建沙盒。

试一试：创建并激活 `virtualenvs`

这个“试一试”展示了如何安装、激活、解除激活 `virtualenv`，以及如何从系统中移除它。

(1) 使用适用于你的系统的命令安装 `virtualenv`：

OSX:

```
brew install virtualenv
```

Linux:

```
apt-get install python-virtualenv
pacman -s install python-virtualenv
```

Windows (powershell users):

```
pip install virtualenv
```

(2) 移动到你将会工作的目录。一些高级用户会在他们的系统上创建一个 `temp_env` 目录并在其中创建 `virtualenv`。如果你有很多 `virtualenv` 要管理，这是一个非常好的工作流程。然而，对于你的目的，你会保持简单。进入目录之后，创建 `virtualenv`：

```
$ cd chapter6
$ virtualenv ch6Ex3
$
```

(3) 如果在创建 `virtualenv` 的目录中做一个有关内容的目录列表，应该会看到一个表示环境名的目录(在这里是 `ch6Ex3`)。你会使用该目录激活你的环境。该目录中包含了所有的安装文件和 Python 代码的目录。想要激活新的 `virtualenv`，只需要添加如下命令：

```
$ source ch6Ex3/bin/activate
(ch6Ex3)$
```

当在一个活跃的 `virtualenv` 中工作时，命令提示符会在命令提示符之前的圆括号中展示 `virtualenv` 的名称。在这里，它是 `(ch6Ex3)$`。

(4) 现在，我们做一个实验。如果你是在第 5 章做这个练习，应该已经通过 `pip install requests` 安装了 `requests`。启动 Python 解释器，查看是否能够使用 `requests`：

```
(ch6Ex3)$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
```

Python shell 的版本是多少？它是 3.4？还是 2.7？如何可以修改它呢？一旦在导入 requests 后按下 Enter/Return 键，就应该会看到一个 ImportError 异常。这表示没有 requests 模块。这是因为尽管你将 requests 导入到系统范围的 Python 中，但是现在还没有使用这个 Python 环境。如果想要在这个 virtualenv 中使用它，就必须重新安装 requests。

(5) 使用 exit() 退出解释器，并在 virtualenv 内 pip install requests:

```
(ch6Ex3)$ python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named requests
>>> exit()
(ch6Ex3)$ pip install requests
Downloading/unpacking requests
  Downloading requests-2.2.1-py2.py3-none-any.whl (625kB): 625kB downloaded
Installing collected packages: requests
Successfully installed requests
Cleaning up...
(ch6Ex3)$
```

在安装包之后，你仍然工作在 virtualenv 中，并且一直会在该环境中，直到你解除激活该环境。

(6) 注意，即使在安装包之后，仍然工作在 virtualenv 中，并且一直会在该环境中，直到解除激活该环境。可以像下面这样不激活你的环境：

```
(ch6Ex3)$ pip install requests
Downloading/unpacking requests
  Downloading requests-2.2.1-py2.py3-none-any.whl (625kB): 625kB downloaded
Installing collected packages: requests
Successfully installed requests
Cleaning up...
(ch6Ex3)$ deactivate
$
```

已经成功安装了 virtualenv，创建了一个新的 virtualenv，为该环境安装了一个包，然后解除激活 virtualenv。如果想要完全移除该环境，该怎么办呢？假如已经完成了那个项目，而且想要移除那些你安装的文件。

(7) 要想移除 virtualenv，需按照步骤移除它创建的目录：

```
(ch6Ex3)$ pip install requests
Downloading/unpacking requests
  Downloading requests-2.2.1-py2.py3-none-any.whl (625kB): 625kB downloaded
Installing collected packages: requests
Successfully installed requests
```

```
Cleaning up...
(ch6Ex3)$ deactivate
$rm -rf ch6Ex3/
$
```

示例说明

virtualenv 为 Python 开发者提供了一种创建有各种版本要求的环境的方式。这有助于保持环境之间的分离，并允许系统为不同的 Python 项目设置沙盒。

有时，在环境中有多个人工作在项目上。当你有一个很长的项目需求列表并且有四个人在不同机器上对项目进行开发时，会发生什么呢？你想要让你的四个组员都一遍又一遍地输入 `pip install <module_name>` 吗？不，你不想。

Virtualenv 有一个非常好的特性。它允许生成一个 `requirements.txt` 文件并把程序需要的包放进该文件中。任何使用包的人简单地输入 `pip install requirements.txt` 就可以安装包需要依赖的所有模块！就是这么简单！

试一试：创建一个 `requirements.txt` 文件来简化添加模块

这个“试一试”展示了如何创建一个假的 `requirements.txt` 文件并用一些常用的包填充它。

(1) 创建一个新的 **virtualenv**：

```
$virtualenv ch6Ex3
$source ch6Ex3/bin/activate
(ch6Ex3)$
```

(2) 编写 `requirements.txt` 文件。在第 6 章目录中，创建 `requirements.txt` 并添加如下行：

```
BeautifulSoup==3.2.0
requests
https://github.com/django/django/tarball/master
```

这些行的内容都是不同的。通常，`requirements.txt` 文件内容都是一致的。但是为了演示的目的，这些行展示了通过 **pip** 安装包的三种最常用的方式。

第一行(`BeautifulSoup==3.2.0`)展示了想要安装 **BeautifulSoup**(一个 Web 抓取库)，但是你想要 3.2.0 版本，所以使用了双等号符号。

第二行(`requests`)安装当前版本。

最后一行(`https://github.com/django/django/tarball/master`)表示想要下载并安装 URL 中提供的包。在这里，会下载并安装整个 Django 目录的 master 分支上可用的 Django 项目(这是一个非常大的文件，所以在下载时需要等待一段时间)。

(3) 保存文件，然后激活 **virtualenv** 并安装在 `requirements.txt` 文件中的包：

```
$ source ch6Ex3/bin/activate
(ch6Ex3)$ pip install -r requirements.txt
```


你应该看到关于下载和安装所提供的这三个包的消息。成功下载和安装包之后，应该会看到一个 `Cleaning up...` 消息，之后是 `virtualenv` 提示符：

```
Successfully installed Django
Cleaning up...
(ch6Ex3)$
```

(4) 启动 Python，并查看那些包是否已安装：

```
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> import BeautifulSoup
>>> import django
>>>
```

如果可以导入这几个包而没有抛出任何错误，那么已经为项目成功安装了所有需要的库。现在可以执行 `rm -rf ch6Ex3 virtualenv`。这会移除 `virtualenv` 和所有已经安装的包，包括非常大的 Django 项目。

示例说明

`requirements.txt` 是 `virtualenv` 的一个特性。它包含了模块正常工作所必需的库。通过 Python 包，开发者就可以使用这个模块。这允许环境的快速设置，这样开发者就可以快速进行开发。

6.6 发布 Python 包

在向外界发布代码时，`__init__.py`（双下划线，`init`，双下划线）文件是非常重要的。对于 Python 项目来说，`__init__.py` 需要位于代码库目录结构的每一层。例如，假设你有一个非常大的代码库，它包含多个 `.py` 文件。首先，你会将一个 `__init__.py` 文件放入目录结构的第一层中：

```
my_package
|---- __init__.py
|---- my_package.py
|---- my_subpackage
|---- __init__.py
|---- my_subpackage.py
```

这告诉 Python 解释器：你想要把这个目录作为一个 Python 包。很酷的一点是：可以保持 `__init__.py` 文件为空，或者可以放一些配置变量在其中。通常，人们会在 `__init__.py` 文件中导入一些模块/库，或其他配置。这些是包函数的基础设置。

当创建 `__init__.py` 文件并导入一些东西时，会发生什么呢？此时 Python 的命名空间是

如何工作的呢？假设在 `my_package/__init__.py` 文件中有如下导入语句：

```
from file import File
```

当想要调用 `my_package.py` 文件中的导入时，可以简单地使用如下命令：

```
from my_package import File
```

`__init__.py` 文件的另一个用途是将所有你想要导入的模块导入到包的命名空间中。为此，需要将 `__all__` 变量赋值为第一层的 `__init__.py` 中的子包：

```
__all__ = ['my_subpackage']
```

这样就可以实现它。当用户从 `my_package import *` 声明时，它会导入 `my_subpackage` 中的所有模块。



注意：你能想象出一种在 Flask 应用中使用 `__init__.py` 文件的方式么？那会让代码更加 Python 化（我们不这样做，因为我们想要向你展示做事情最原始的方式，这样你就可以在添加更多步骤和层次之前理解这些事情的内在工作原理）。

现在，你已经编写了代码并且已将 `__init__.py` 文件放在了正确的地方，如果想要向外界发布代码会如何呢？如果想要在其他机器上输入 `pip install <package_name>`，安装这个模块该如何呢？

PIP 和 PYPI

在本书中，已经使用过 `pip` 安装第三方库和模块。但什么是 `pip`？它是如何工作的呢？

`pip` 是 Python 包安装程序。它安装 `PyPI`（发音为“pie-P-I”，而不是“pie-pie”）中的包。`PyPI` 是 Python 包索引。它的另一个名字“The Cheese Shop”（另一个 Monty Python 引用）也被更多有经验的开发者所熟知。在 `PyPI` 中，你可以上传自己的 Python 包，这样可以通过 `pip install <package_name>` 获得它们。有时，人们只会将自己的包上传到 `PyPI`，因为这样他们可以更轻松地在多个机器上安装那些包。有时，人们上传他们的包是因为他们希望可以帮助其他人。

想要找到更多的包索引或搜索索引，可以访问 <http://pypi.python.org/pypi>。这是 `PyPI` 的主页，其中包含了你需要的所有信息。

如果想要将自己的包上传到 `PyPI`，则需要在 `PyPI` 上注册，然后按照主页上的链接的教程来做。这确实十分简单。先注册，上传你的包，稍后就可以通过 `pip install <package_name>` 使用这个包了。

要记住的是，当把一个包上传到 `PyPI` 时，就说明任何人都可以下载并使用。这是为什么要时时刻刻练习好的、Python 化的编程十分重要的原因。你永远不知道什么时候会有人

下载并使用你的模块，并且你想要让他们以最小的代价来使用你的作品。

6.7 本章小结

我们已经了解了一些为 Python 项目进行基本的测试和打包的方法。你现在应该对大多数 Python 包和模块/库是如何架构和创建的有了清晰的理解。对于读者来说，一个好的练习方法是回到本书的一开始并使用在本章所学的概念做一遍练习。你能用测试驱动的方式重写第 3 章的代码吗？你能打包第 5 章的 Flask 应用并把它发送到另一台计算机上运行和开发么？你应该试试这些东西，这样你对 Python 包的所有部分是如何协同工作的就有了一个清晰的理解。

练习

- 1. 在本章的 zip 文件中，打开文件 `markets.py` 并编写 `doctest` 字符串来测试文件中函数的返回值。你能想出为什么在这个代码中，一个简单的 `doctest` 字符串对于将来维护代码如此有用吗？
- 2. 为函数编写一个 `unittest`，它接受一个字符串并返回这个字符串的倒序。确认测试会失败，因为你还没有编写函数来测试。
- 3. 为你的单元测试编写一个函数。它接受一个字符串并返回这个字符串的倒序。现在，对这个函数进行单元测试并修改函数直到它通过。

本章所学知识

主 题	关 键 概 念
单元测试	通常是指被写在一个单独测试脚本中的函数。它导入将要测试的代码并测试导入代码中的每个函数
Virtualenv	第三方软件。它允许开发者使用自定义的 Python 版本和 Python 库/模块为 Python 开发创建系统沙盒
TDD(测试驱动开发)	一种开发风格。一个人要先写测试，这些测试会失败，然后编写实际的函数代码让测试通过。在这种风格下，测试优先是驱动整个开发周期的动力
Pdb	Python 调试器，Python 中的一个交互式调试模块

7.1 使用 Python 绘图

Python 中有许多绘图和图形处理工具。这些工具包括从简单的绘图(比如 `matplotlib`)到高级专业的数据可视化和分析(比如 `matplotlib` 和 `pillow`)。下面概述了每个工具的功能和应用场景。

第 7 章

探索 Python 前沿技术

本章主要内容:

- Python 在专业领域中的应用
- 用于专业领域的第三方包
- 如何对 Python 的发展做出贡献

从 wrox.com 下载本章代码

本章并没有 wrox 可供下载的代码,但是这里讨论的各种解决方案都有可以下载的包。其中许多包都可以通过 Python 包索引(Python Package Index, PyPI)安装或者从包的主网站下载二进制安装器。每个部分都会提供包下载地址的有关信息。

在本书的之前章节中,你已经看到如何用 Python 与操作系统和其他程序交互,如何用它通过平面文件和数据库来管理数据,以及如何创建桌面和基于 Web 的应用。你也看到了一些可以有效、可靠地创建更大型程序的技术和工具。

在最后一章中,你会看到 Python 如何支持更多编程领域。会学习各种各样的框架、包、库和甚至是独特的 Python 分发版本。这些版本被开发来支持特殊兴趣领域,比如科学和语言处理。你也会看到有哪些专业工具和包支持一些特殊的应用类型以及它们的特殊需求。最后,将看到如何为 Python 社区做出贡献来帮助 Python 变得更好。

7.1 使用 Python 绘图

Python 中有许多绘图和图形处理工具。这些工具包括从简单的绘图库(比如 turtle)到高度专业化的模块和框架(比如 matplotlib 和 Pillow)。下面描述了每个工具的功能和应用领域。

7.1.1 使用 turtle graphics

在代码中绘图的最简单方式是使用 turtle graphics。turtle graphics 最初是一种使用程序设计语言 Logo 绘图的方式。它的思想是向一个机器人设备——turtle——发送指向性命令，然后这个设备会产生图片。这个概念非常流行。现在，大多数语言都在某种程度上支持 turtle graphics。Python 提供了 turtle 模块。默认情况下，模块会使用 Tkinter 在一个小的弹出窗口中展示图形。在应用初始化 turtle 系统时，可以指定一个 Tk 画布对象(下部分会有更多关于画布对象的详情)。也可以在交互式提示符中使用模块来试试这个系统。官方文档详细描述了它的函数和方法。通过示范教程，可以很好地理解它的功能。在操作系统命令提示符中输入如下命令：

```
python -m turtledemo
```

这会启动一个带有示例菜单的窗口。可以启动和停止这些示例，也可以展示示例的代码。这样可以看到如何在程序中实现同样的效果。

7.1.2 使用 GUI Canvas 对象

大多数 GUI 框架都包含画布(canvas)对象。画布是屏幕上的一块区域。可以在里面绘制线和图形、添加图片，甚至插入文本。Tkinter Canvas 对象非常经典，它支持弧形、椭圆形(包括圆形)、线、长方形、多边形、文本、图片，甚至还包括窗口(这样可以在画布中嵌入一个小部件)。以下这个画布小程序显示了一个红圈：

```
>>> import Tkinter as tk
>>> top = tk.Tk()
>>> c = tk.Canvas(top, width=50, height=50)
>>> c.pack()
>>> c.create_oval(10,10,40,40,outline='red',fill='red')
1
>>> top.mainloop()
```

Canvas 类包含许多方法，可以用来创建复杂的图形程序。

这还只是在抽象的低层。如果想要达到一个更高的层次，可以使用其他库，比如之前讨论的 turtle，或一些将要讨论的更加奇特的第三方选项。

7.1.3 绘制数据

Python 中最流行的绘制数据工具是 matplotlib。可以在 <http://matplotlib.org/> 找到它并且从 PyPI 上下载它。它也是之后本章讨论的 scipy 包的一部分。该网站提供了许多示例和教程的链接。

matplotlib 与其他 scipy 包紧密相连。因此，如果只想要一个简单的图形，它还是很令人敬畏的。PyPI 上还有其他轻量级的包。它们试图通过提供易用的绘图库来解决这个问题。但是对于正式的绘图，matplotlib 是最好的选择。

7.1.4 使用 imghdr

如果你的兴趣在于图片而不是数据，`imghdr` 模块则提供了更加有用的帮助。它可以帮助你决定图片文件的类型。这个模块是标准库的一部分，非常易用。它通过测试文件的数据内容来决定图片类型，而非依赖文件的扩展名。

这个模块只包含一个函数 `what()`。它接受文件名或文件流作为参数并返回图片类型。模块支持大多数常见的文件类型，但是可以通过添加自定义测试函数来处理其他图片类型。

7.1.5 Pillow 简介

很多年来，Python 中操纵图片的标准解决方案是 Python Imaging Library(PIL)。PIL 还没有被移植到 Python3 中。有一个基于 PIL 的库 Pillow，它添加了一些新特性。

Pillow 的主页是 <http://pillow.readthedocs.org>。可以通过 PyPI 安装它。

Pillow 是基于 Image 类的，可以被打开和保存。如下程序将 JPEG 文件转换为 PNG 文件：

```
>>> from PIL import Image
>>> Image.open('foo.jpg').save('foo.png')
```

也可以使用 Image 对象获取图片的信息，比如它的大小。它还有许多更加强大的功能。例如，可以通过翻转或旋转来转置图片，以及改变大小和应用滤镜。Pillow 就像程序驱动的图片编辑程序。

7.1.6 试试 ImageMagick

ImageMagick 是一个类似于 Pillow 的工具。但是它基于同名的命令行工具包。命令行的网站是 <http://www.imagemagick.org/>。

PyPI 上有一个 Python 包 wand，它使用 ctypes 包装了大量 ImageMagick 的功能。它的网站是 <http://docs.wand-py.org>。网站上的文档包括一个用户指南和一些参考。

类似于之前部分中的 Pillow 示例，如下是一个图片转换程序：

```
>>> from wand.image import Image
>>> with Image(filename='foo.jpg') as img:
...     img.format = 'png'
...     img.save(filename='foo.png')
>>>
```

还有许多其他 Python 中可用的图形模块和包。PyPI 搜索工具和 Web 搜索引擎可以搜索到更多的示例。本章已经提到的工具应该覆盖了大部分应用场景，但是也可以试试其他工具。

7.2 使用 Python 辅助科学

Python 在科学和数学社区中的应用历史很悠久。因此，创建了许多模块和包来满足社区的专业化需求。在了解第三方选择之前，你应该首先了解 Python 提供的内置支持。

Python 的原生类型为科学计算提供了很多支持。特别是，Python 高效而无大小限制的整型类非常适合大量而长序列的计算。Python 的浮点类型与其他语言很像。但可以选择使用十进制或分数。这可以减少因为舍入而产生的错误。最后，Python 是为数不多的支持复数或虚数的语言之一。这种数字类型在科学和工程中有广泛的应用。

当然，除了多种数据类型，还需要支持这些数据的操作。同样，Python 通过标准库模块(比如 `math`、`cmath` 和 `statistics`)补充了内置操作。这些模块提供了更多选项。这些模块，比如 `collections` 模块，支持更加奇特的数据类型(例如命名元组、有序字典和链映射)。链映射是链接多个映射的有效方式。

尽管这些工具非常强大，但是它们仍然不支持详细的科学分析。这也成为特殊第三方库的舞台。这些库中的主力是 SciPy 包。

7.2.1 SciPy 简介

SciPy 有很长的历史。它是从几个独立的开发流中演化而来的。这些开发流慢慢形成了一个强大的集成体。SciPy 项目集成了 6 个单独的库，它们形成了一个集成工具栈。SciPy 的网站是 <http://www.scipy.org/>。

这 6 个库分别是：

- **NumPy**: Python 中最早的数学包之一，并且是许多其他包的基础。Numpy 包含一组适用于数字分析和模拟的数据类型和操作，包括 N-维数组对象、线性代数、傅里叶变换和各种随机数生成器。此外，NumPy 还可以提供用 Fortran 和 C 编写的科学和数学工具。
- **SciPy**: 项目命名的包。它包括集成、信号处理、稀疏数据结构和许多用于特殊目的的函数。
- **Matplotlib**: 我们已经在本章的绘制数据部分讨论过它。它可以生成高质量的图，适用于发布，还提供大量布局控制和标签等。它的目的是与商业包(比如 Mathematica 和 MATLAB)竞争。它也支持数个 GUI 工具包。这些工具包可以用来创建拥有丰富图形的桌面应用。
- **SymPy**: 专为符号数学设计。它使用表达式(例如 `sqrt(2)`)作为答案中的符号，而不是数字结果。这对没有数学背景的人来说可能有些奇怪，但是对于数学家来说它是一个标准工具。使用它可以不必面对传统浮点数表示法因为舍入而产生的错误。可以把它当作纯正数学而不是算术的工具。对于前者，结果是符号化的，而后者是数字。可以使用它来解决积分和微分问题、贝塞尔函数、特征值等。SymPy 有一个可以输入表达式的交互式 shell 提示符。还有可以导入到应用中的包。

- **Pandas:** 一个数据分析工具包。在第3章,你接触过用来与R统计分析语言交互的 rpy。Pandas 是 R 环境的纯正 Python 替代方案。在本书成文时, Pandas 在这点上没有 R 那么强大,但是它是一个逐步成长的项目。此外,它还集成了比 rpy 更好的其他 SciPy 包。如果只对统计感兴趣,请使用 rpy。但是如果想要更加集成地分析 workflow 或使用其他 SciPy 元素,那么请仔细看看 Pandas 吧。
- **IPython:** 这不是专门针对科学用户的。它是标准 Python 交互式解释器的强大替代品。它把传统的 `>>>` 提示符替换为 `In[n]:`, `n` 是命令的序号。输出行是以 `Out[n]:` 开头的,这里 `n` 对应 `In[]` 提示符里的值。除了所有常用的 Python 语言, IPython 还添加了多个新特性,例如 `help()` 函数的快捷键。你也在操作系统命令前添加感叹号(!) 来执行它。但 IPython 不仅仅是交互式提示符,它还包括一个笔记本功能。在 IPython 中,整个会话可以被保存和取回。因此,可以在多个项目上工作。在完成时,可以保存每个项目的状态,然后完美地恢复项目状态,并继续使用所有历史记录。IPython 也与其他 SciPy 库兼容,包括 matplotlib 和 SymPy,还可以展示图形或符号表达式。可以在 IPython 网站 <http://ipython.org/> 中的完整文档找到关于笔记本的示例。matplotlib、SymPy 和 IPython 的组合为商业包(比如 MATLAB 或 Mathematica)提供了一个强大的备选方案。

最后, SciKit 横幅包括多个 SciPy 附加项。这些覆盖了航空工程、语音、图片、视频处理、环境科学等领域。

7.2.2 使用 Python 辅助生物学

近年来,一个前沿科学领域是生物科学和 DNA 分析。bioPython 包已经被开发来满足这个需求(<http://biopython.org/>)。这个包支持读取和写入大部分生物信息学的标准文件。它还有一个用于分析 DNA 序列的 Sequence 类。

除了 bioPython,还有一些其他模块。可以使用搜索引擎找到它们。在试图实现模块前,最好先检查是否别人已经帮你完成了工作。

7.2.3 使用 GIS

随着卫星导航和移动电子地图软件的爆发,地理信息系统(GIS)领域正在急剧升温。ArcGIS (<https://www.arcgis.com>)是专业的地理信息处理标准工具集。Python 通过 ArcPy 支持 ArcGIS。它的目的是帮助你访问地理信息处理工具。它还包含一些额外的函数和类,可以帮你快速轻松地创建简单或复杂的工作流模块。

还有一个大问题。在本书成文时, ArcPy 只支持 Python v2.7,并不支持 Python v3。但 ArcPy 是当前 GIS 处理的最佳选择。

7.2.4 处理语言

多年以来,人类语言的学习和把自然语言处理为数据已经发展成一个研究领域,并已

经有了长足的发展。随着计算能力的增强，自然语言处理已经开始出现在主流项目中。

Python 通过 Natural Language ToolKit 来支持这个领域。它的主页网址为 <http://www.nltk.org/>。这个工具包提供了多个专业工具，并帮助开发者解析并标记文本，分析它的结构，并对它分类。可以从 PyPI 找到并安装 NLTK。

7.2.5 综述

尽管所有之前讨论的包都非常强大，但是使用普通安装工具把它们安装到标准 Python 发布版本中是一个非常繁琐的过程。幸运的是，可以选择使用 Anaconda 和 Enthought Canopy。这些 Python 发布版本已经带有所有你可能需要的科学工具。除了已经讨论过的 SciPy 和 NLTK 框架，Anaconda 有超过 100 个内置的其他专业包。这个版本是由 Continuum Analytics 发布的。它基于商业版本的基础上提供其他包。Canopy 也非常相似。它提供一个免费的基础版本以及增强的商业版本。

Anaconda 可以被安装在 Windows、MacOS 或 Linux 上，并且不与现有的 Python 安装版本冲突。在本书成文时，Anaconda 支持 Python 2.6 到 3.4。Anaconda 的主页网址为 <https://store.continuum.io/cshop/anaconda/>。

Canopy 也同样可以用在所有主流平台上，它的主页网址为 <https://www.enthought.com/products/canopy/>。

7.3 使用 Python 开发游戏

在第 4 章“创建桌面应用”中，已经看到如何使用 Python 创建基础游戏。在该章中，你基于经典的 tic-tac-toe 游戏创建了多个版本。然而，大多数游戏玩家在今天会期待一些更加令人兴奋和动态的东西。Python 可以支持许多类型的游戏，并且全面支持随机数生成。这是任何游戏的核心部分之一。它是基于标准库的 random 模块实现的。这个模块可以模拟投掷骰子、从多个选项中随机选择，或者产生一个各种格式的随机数。

7.3.1 增强 PyGame 经验

如果想要拥有使用多媒体的丰富游戏经验，第三方包 PyGame 是一个好的开始。它提供了一组包含 Simple DirectMedia Layer(SDL)的模块。它可以帮助程序访问语音、键盘、游戏杆和图形硬件。它也是跨平台的，所以 PyGame 能在大多数流行操作系统上工作。它是模块化的，所以只需要使用需要的部分，这会让你的代码保持很小。

它的网站为 <http://www.pygame.org>。其中包含了许多示例程序，以及很多教程。PyGame 有一个活跃的可以提供帮助和支持的用户社区。还有多本使用 Python 和 PyGame 创建游戏的书籍。

7.3.2 探索其他选项

PyGame 并不是唯一一个专注于游戏的库。你还有很多其他选择，例如 Pyganim。它是一个基于 PyGame 的精灵动画模块，但是更易用。Albow 是一个 GUI 工具包，专门用于使用 PyGame 创建游戏。许多其他包都是基于 PyGame 编写的。这也证明了它作为基础游戏框架的流行度。

当然，并不是必须要使用 PyGame。还有其他访问低层硬件和库的包，如 PyOpenGL，顾名思义，该包提供了对 OpenGL 库的访问。

另一个游戏特性是用于建模真实世界物理对象行为的物理引擎。Python 也有工具来支持它，比如 2D 模型 pymunk。Panda 3d 和 Python Computer Graphics Kit(cgkit)提供了 3D 支持。

除了图形，大部分游戏需要声音。对于此，可以使用标准库中的内置模块，如 aifc、wave 和 sunau。winsound 模块提供了对 Windows 声音设施的底层访问。基于这些低层库，游戏社区已经创建了几个包来帮助产生配合动作的适度兴奋的声音。

还有许多其他库。实际上，游戏开发者面临的选择让人眼花缭乱。可以在 <https://wiki.python.org/moin/PythonGameLibraries> 找到有用的总结。

7.4 进入电影领域

Python 在电影业务上有很长的历史。有几个计算机成像(Computer-Generated Imagery, CGI)工具使用 Python 作为脚本引擎。几个有名的电影使用 Python 做一些幕后的生产工作。为了支持这个，各种包已经被开发出来并且对更多的观众可用，包括 Nuke、Maya 和 Blender。许多这些包都基于本章前面使用 Python 开发游戏部分的 cgkit 包。这意味着你在创建和编辑视频作品时有很多使用 Python 的选择。

计算机图形套件

cgkit 包提供了一组创建 3D 场景所需的底层类型和操作，它也提供一个渲染引擎。如果需要的话，结果可以被显示在其他渲染引擎中。cgkit 包含 Pixar RenderMan API 接口。cgkit 也包括 Maya 插件。这可以让 Maya(参阅下一节内容)与 Python 之间进行交互。

尽管有可用的教程和引用文档，但是仍然需要 3D 计算机图形原理的基础知识，这在使用 3D 模型应用时可以获得。

版本 2 的 cgkit 是在 2013 年初发布的。cgkit 支持 Python 2 和 3。这个包非常稳定，几乎没有任何新的开发在进行。

cgkit 的主页网址为 <http://cgkit.sourceforge.net/introduction.html>。

1. 建模和动画

许多工具可以用于视频图片的数字合成。在这里讨论的工具是一个子集。所有这些都

有一定程度上的 Python 集成。

NUKE 产品家族的目标是针对专业的视频图形市场。它是一个集成了 Python 的商业合成工具，但是在匹配目标观众时需要付出一些代价。它有一个免费试用版，是非商用的、功能受限的个人学习版本。可以在 NUKE 主页 <http://www.thefoundry.co.uk/products/nuke-product-family/> 上找到它。NUKE 使用 Python 2.7。

Maya 是另一个 3D 动画合成工具。它也是一个与 NUKE 竞争的商业产品，并且也提供一个免费试用版本。它可以使用 Python 脚本化，并且可以将 Maya 动画集成到 Python 程序中。Maya/Python 集成是在之前描述的 cgkit 库的一部分。Maya 主页网址为 <http://www.autodesk.co.uk/products/autodesk-maya/overview>。

Blender 也是一个动画合成包，但它是开源免费的。这让它在消费者市场更加受欢迎。它也使用 Python 作为它的脚本引擎。主页网址为 <http://www.blender.org/features/>。Blender 使用 Python 3。

2. 图片处理

对于处理摄影图片，本章在 7.1 节“使用 Python 绘图”所讨论的许多解决方案都适用。Pillow 和 ImageMagick 都是操纵摄影图片非常有效的工具。它们能够裁剪、改变曝光度等。此外，SciPy 中的库包含了处理图片的库，例如 Gaussian blur。

也可以在 PyPI 中找到许多模块来执行特定任务，比如改变大小和裁剪图片。模块 psd-tools 可以读取 Adobe Photoshop .psd 文件。还有像 pyexif 一样的模块，它在底层使用 exiftool 命令行处理 EXIF 元数据。

最后一种照片工具是在线媒体管理器。一些模块可以帮助你与各种网站传输图片，比如 Picasa、Flickr、Facebook 和 Twitter。其中一个模块是 picasa-downloader。遗憾的是，许多模块仅在 Python v2 中可用，并且大部分还是在底层使用 Pillow 或 ImageMagick 工具。

3. 处理语音

你已经听说过标准库中的内置模块：aifc、wave、sunau 和 winsound。这些只适用于应用或非游戏声音应用。

SciPy 中的各种包也可以用于处理声音文件，尤其结合一些 SciKit 附加项。这非常适用于分析声音内容或绘制信号波形。

有一个有用的 Python wiki，它列出了许多面向声音和音乐的项目。网址是 <https://wiki.python.org/moin/PythonInMusic>。

7.5 与其他语言集成

至今，你一直使用的 Python 版本都是用 C 语言创建的。它通常被称为 CPython。Python 语言还有使用其他语言实现的版本。这些非 C 解释器帮助 Python 与其他语言协作。两个

最有名的其他 Python 版本是用 Java 编写的 Jython 和为微软的 .NET 环境实现的 IronPython。第三种版本是 Cython。它并不是严格意义上的 Python 实现，但是它与 Python 非常密切相关。它可以被编译成 C，从而提升性能，并同时提供与 Python 语言类似的开发速度。最后，在 Tkinter 包中可以访问 Tcl/Tk 代码。

7.5.1 Jython

Python 的 Java 实现版本为 Java 程序员提供了许多优势，他们一直寻求在交互式环境中测试 Java 类或创建原型解决方案。如果需要的话，原型可以被转换为完整的 Java 代码。

Jython 发行版本包括一个解释器和一个编译器。解释器拥有熟悉的交互式提示符，以及直接运行脚本的库。除了可以导入 Python 模块(包括许多常用的标准库模块)，Jython 还可以导入 Java 库。这让 Java 类在 Python 解释器中用起来就像普通的 Python 类。这也可以在 Jython 提示符中交互式地练习和测试新的 Java 类。Jython 也可以混合 Java 和 Python 代码来支持解决方案的动态原型化。解释器也可以在更大项目或原型中用同样的功能来运行脚本文件。

编译器将 Jython 代码(纯粹的 Python 代码或 Java 和 Python 代码的混合)编译成一个 .java 文件。这是一个可以原型化新类的强大工具，因为它们可以用 Python 编写和开发、编译，然后包含在 Java 代码中。成功之后，Python 版本可以被无缝地替换为纯粹的 Java 版本。

Jython 的弱点在于它产生的代码会比纯粹 Java 慢一些，也更耗内存。这主要是因为编译器在输出文件中高效地嵌入了 Python 解释器。

在本书成文时，尽管移植到版本 3 的工作仍在进行，但 Jython 的版本仍然是 2.7。

7.5.2 IronPython

IronPython 是为 Microsoft .NET 框架编写的 Python 版本。.NET 不是一个单独的语言系统，它实际上依赖一个公共的字节码。很多语言都可以被编译成字节码。这样产生的模块可以在多个语言间共享。因此，用 IronPython 编写的代码可以导入到 C#、C++、Visual Basic 和多个其他与 .NET 兼容的语言编写的模块中。类似的，IronPython 模块可以被任何上述语言导入。对于 .NET 平台开发者来说，IronPython 是非常有吸引力的。

好消息是，一个名为 Mono 的开源 .NET 版本已经可以运行在 Linux、Mac OS X 和许多其他平台上，包括主流计算机和游戏控制台。它维持了与 Microsoft .NET 二进制级别的兼容性(另一方面，一个轻量级的限制版本 Mono Touch 可以运行在 iOS 和 Android 上。它可以被用来创建智能手机应用)。由于 .NET 已经成为 Microsoft Windows 应用开发的业界标准，Python 在框架上的可用性对 Python 程序员有很大帮助。

IronPython 支持大部分标准 Python 库以及 .NET 模块系统。在 .NET 中，模块被称为程序集。但是导入 IronPython 的方式与导入标准 Python 模块的方式完全一致。由于 Python 使用的动态类型和本质上更加静态的 .NET 类型系统存在一些差异，IronPython 仍然存在着一些问题。然而这些问题可以使用 IronPython 内置的帮助功能解决。完整文档在 IronPython

文档网站上: <http://ironpython.net/documentation/>。

在本书成文时, IronPython 与 Python 2.7 兼容。还有一个开发 Python 3 的项目正在进行。有一组工具有助于在 Microsoft Visual Studio IDE 中进行 IronPython 开发, 其中 Microsoft VisualStudio IDE 是 .NET 开发的默认 IDE。

7.5.3 Cython

Cython 与其他语言版本有很大的区别。实际上, 它与 Python 是不同的语言, 但是可以与之高度兼容。它把自己描述为 Python 的超集。这意味着 Python 开发者可以轻松的学习 Cython 并利用它的特殊功能。

那么 Cython 有哪些功能会让你想要使用它呢? 简而言之, 就是速度。Cython 是一个产生 C 代码的编译器, C 代码可以被编译成原生的机器码并因此拥有比同等 Python 代码更快的执行效率。编译好的代码可以像任何其他模块那样被导入到 Python 中。这样可以同时拥有便捷的 Python 开发以及 C 语言级别的执行速度。

Cython 的扩展主要与原生 C 代码交互。在这一点上, 它与第 2 章的 2.4 节中讨论过的 ctypes 模块的帮助功能非常类似。



注意: 如果 Python 代码导入了一个已经用 C 编写的模块并执行了它的函数, 那么把程序再转换为 Cython 将会有一些效率上的影响。但如果代码包含大量纯粹的 Python 代码, 这个差异会很大。

Cython 不只是一个 Python 到 C 的翻译器, 它可能是众多 Python 开发者最易用的工具。Cython 网站是 <http://cython.org/>。

7.5.4 Tcl/Tk

tkinter 和 tix GUI 模块是基于 Tcl/Tk 和 Tix 工具包创建的。因此, 它们可以在 Python 中使用内嵌的 tk 对象方法来执行 Tcl 代码: `self.tk.call()`。

这个方法是 tix 模块创建的关键。如果查看 `tix.py` 代码, 你会发现许多类似于 Notebook 类中的方法定义:

```
def raised(self):  
    return self.tk.call(self._w, 'raised')
```

如你所见, 这个方法只是调用底层 Tix 小部件(`self._w`)的一层封装。如果熟悉 Tcl/Tk 和 Tix, 可以非常轻松地扩展现有 Python 小部件来使用一些在其他情况下不能使用的 Tcl 特性。`call()`的替代方法是 `eval()`。它会将输入字符串当作 Tcl 表达式。

但是, 集成并不局限于访问 GUI 小部件。可以向 `eval()`方法传递任意 Tcl 代码, 并让它用内置的 Tcl 解释器执行。这也包括 Tcl 模块的导入。它可以提供 Python 没有的特性。

当然，在熟练使用它之前，需要充分了解 Tcl！这是一个基本的 Hello World 脚本。它可以运行在任何使用标准输出流的命令行中。

```
>>> import tkinter
>>> tcl = tkinter.Tcl()
>>> tcl.eval(''
... puts "Hello world"
... '')
Hello world
''
>>>
```

三引号的内部可以是任意 Tcl 脚本。

7.6 进入物理领域

近些年来，越来越多的业余爱好者对物理设备编程感兴趣。低成本、高度灵活的产品，比如 Arduino 微控制器卡和 RaspberryPi 单板计算机(SBC)的成本已经足够低了。同时，Python 是 RaspberryPi 的推荐语言之一。

并不是必须要花钱买硬件来连接 Python 和外面世界。大多数计算机仍然拥有运行 RS232 协议的序列端口。可以通过这个端口连接到各种外设并使用 Python 访问它们。同样的，可以用 MIDI 接口访问音乐设备，甚至可以在合适的位置用正确的模块来操纵普遍存在的 USB 接口。

在任何类型的集成中，需要理解连接的两端。如果不熟悉想要连接的物理设备，你会很被动。将 Python 与任何东西集成在一起的第一步是找出目标设备是如何与世界交互的。只有这样，下面部分讨论的库才会对你有所帮助。

7.6.1 serial 选项介绍

可以使用 PySerial 项目发布的 serial 模块来访问计算机上的 RS232 通信端口(或任何其他序列端口，包括旧样式的 PS-2 鼠标或笔)。它的网址是 <http://pyserial.sourceforge.net/>。其中有全面的文档和示例。

这个模块支持多个操作系统，包括 Windows 和 Linux，以及 Jython 和 IronPython。它可以通过 PyPI 安装，或在大部分发布版本中作为一个 Linux 包，或在 Windows 中作为一个二进制安装包。它可以在 Python 2 和 3 中工作。

PyUSB 库可以访问 USB。它是用 Python 编写的，并在内部使用 ctypes 访问底层代码。可以在 <https://github.com/walac/pyusb> 上找到 PyUSB。虽然网站上的内容有些少，但是也包含了带有几个示例的教程，它假定你已经理解 USB 如何与函数交互。

PyUSB 可以从 GitHub 仓库获得，但是通过 PyPI 安装会更加轻松。

7.6.2 RaspberryPi 编程

RaspberryPi 是一个单板计算机。它大概与信用卡同样大小，并且售价很低。它本来是用于培养计算机技术与编程兴趣的。它不仅在这个目标上获得了巨大成功，也成功地成为一个业余爱好者工具。许多拥有者利用它的小身材把计算机创建成一些应用的小型物理平台。这些应用包括天气预报、安全系统、机器人实验和车辆控制。

RaspberryPi 可以运行在多个 Linux 上，默认是 Raspbian Linux。它包括完整的 Python，并且 Python 是用户编程的推荐语言。如果使用基本配置，它可以被当成任何其他计算机来使用。确实，如果连接上显示器、鼠标和键盘，它就像任何 Linux 个人计算机一样，尽管它拥有最差的计算资源。如果作为一个自定义项目的控制系统，编程环境可能会使用一些外围访问技术，尤其是使用 PyUSB 访问内置的 USB 端口。

RaspberryPi 的主页网址为 <http://www.raspberrypi.org/>。

有一个免费的社区在线杂志 The MagPi。爱好者可以在里面分享经验、小技巧和项目。此外，已经有多本关于这个主题的书籍出版了，还有定期会议和本地用户组。

7.6.3 与 Arduino 对话

在很多方面，Arduino 产品是 RaspberryPi 的补充。RaspberryPi 的目标是教授计算和编程，而 Arduino 的目标则直接是电子爱好者。它可以对接口进行模拟和数字实验。Arduino 上各种各样的微控制器电路板有很多输入和输出的配置。通常，它们包含一个 USB 接口、几个模拟输入引脚，以及一些数字 I/O 引脚，因此允许用户附加各种外部设备。

附件还可以扩展可以连接的设备类型。还包括一个用 C 编写的 Wiring 代码库。它能够访问各种端口。还有 IDE 帮助用户编写代码并提供一键上传到面板的机制。Arduino 的主页网址是 <http://www.arduino.cc/>。

尽管通常 Arduino 处理器必须下载一个二进制应用，但是它也可以通过连接到控制计算机(比如 RaspberryPi)来控制。这可以使用 Python 通过 USB 或 serial 端口向 Arduino 发送指令。有一个辅助这个过程的库 pyfirmata，可以从 PyPI 安装它。有一个网站深入讨论了它。网站通过多个示例展示了它能做些什么。网站网址为 <http://playground.arduino.cc/interfacing/python>。

7.6.4 探索其他选项

RaspberryPi 和 Arduino 项目的流行已经产生了多个相互竞争的产品。在这些产品中，很多只是其他产品的简单低成本克隆，尤其是 Arduino。但其他产品是真正的替代品。它们拥有轻微不同的目的或想法，例如创建尽可能小的面板。在大多数情况下，Python 可以通过 serial 连接或标准 Python 模块的网络连接访问面板。

你钟爱的搜索引擎应该能够找到许多候选方案。你应该了解提供接口的本质和如何完成编程。一些 Arduino 克隆需要在真正的 Arduino 上对芯片进行编程，然后传输到最终设备的克隆板上进行安装。

7.7 创建 Python

Python 程序员的一个特殊兴趣领域就是 Python 本身。Python 拥有非常开放的架构。它的很多特性允许程序员探索解释器的内部工作原理，以及程序内部的数据结构。作为开源项目，Python 的发展是社区的事情，而且每个使用 Python 的人都可以从它的发展中获得收获。如果认为有些 Python 功能有问题或者可以被改善，有流程来修复。如果想要添加一个功能，也有流程来添加。

如果想要加入 Python 开发，无论是解决个人问题还是反馈社区，都能够以多种方式开始。可以在 <https://docs.python.org/devguide/index.html> 上找到所有选项的详细介绍。

7.7.1 修复 bug

可能加入 Python 的最明显的方式是修复 bug。报告 bug 很有用，但提供修复则更好。有一个官方的 bug 追踪应用。在提交 bug 之前，应该查看它是否已经被报告以及为了修复它正在做哪些努力。如果它是一个新 bug，可以填写一个报告(也可以提供修复方法)。

一旦 bug 被上报，追踪器就支持一个会话模式。在这里，用户可以建议修复方法、评论补丁等。bug 追踪器的主页网址为 <https://docs.python.org/3/bugs.html>。

7.7.2 文档化

在大多数开源项目中，写代码的人要比写文档的人更容易找到。Python 也不例外。尽管官方文档对于一个开源项目来说已经很好了，但是它有很多地方还很粗糙，甚至有一些地方是完全缺失的(在第 4 章看到的 `tix` 模块就是这样。几个 `tix` 控件虽然可用但在官方文档中没有描述)。

可以尝试自愿去为一些 Python 不健全的地方添加文档。而这也非常温柔地介绍了开源社区。文档问题被报告在标准 Python bug 追踪器上。可以使用它来提交 bug 和修复建议。如果想要更加深入地投入其中，可以订阅 docs@python.org 邮件列表。可以在网站 <https://docs.python.org/devguide/docquality.html> 上找到更多详细信息。

文档实际上是用特制的文档 Sphinx 处理器生成的。Sphinx 内容是使用 reStructuredText(reST)创建的。这是一个轻量级的标注系统，类似于许多 wiki 页使用的那些(Python Docutils 项目为处理 reST 文件提供了底层的工具集)。Sphinx 网站是 <http://sphinx-doc.org/index.html>。

7.7.3 测试

除非计划加入核心 Python 开发，否则对测试最好的贡献就是下载并使用早期测试版本。然后可以像往常一样使用 bug 追踪器来报告 bug。

7.7.4 添加特性

如果感兴趣的地方不是 bug 而是缺失的或不完整的功能,不管是 Python 语言本身还是模块,你都应该考虑将想法提交到 python-ideas 邮件列表。可以在 <https://mail.python.org/mailman/listinfo/python-ideas> 上注册这个列表。这个列表可以让你的想法在更大的社区中讨论。如果想法不错,可能会邀请你提交一个 Python 增强型提案或 PEP。

必须说明的是,你获得模块更改批准的机会要远大于修改一个核心语言功能。但是语言修改有时也会发生。好想法是值得尝试的。

7.7.5 参加会议

另一个参与 Python 社区更加简单的方式是参加各种各样的本地用户组和每年举行的会议。这让你有机会学习 Python 和它的底层、了解基于 Python 的成功项目,并理所当然地用 Python 展示你自己的经验。

几个年度国际 Python 会议一年举行一次。还有一些小的事件,或者集中于一个本地区域或者一个特定的兴趣组。详细信息通常会在各种邮件列表中在线公布。列表的维护在网站 <https://wiki.python.org/moin/PythonEvents> 上。

7.8 本章小结

在本章中,你了解了 Python 更宽泛的内容,学习了 Python 标准库一些未覆盖的但在更大社区中提供了广泛支持的专业领域。

你看到了如何使用核心 Python 模块以及各种第三方库生成和操纵图形,特别是 Pillow 和 ImageMagick。

许多第三方库可以用于科学领域,而很多这些库的基础就是 SciPy 包含的包和工具。类似 Anaconda 的发布版本可以让安装这些包变得轻松很多。

对于游戏编程,你发现 PyGame 包提供了底层图形和多媒体访问。第三方计算引擎也有助于你开发现实游戏。

Python 有商业产品负责视频处理和影片世界。Blender 为你提供了一个开源选择来创建视频作品。

Python 可以通过多种形式与其他语言结合在一起。Jython 提供了与 Java 的双向集成,而 IronPython 集成了 Microsoft 的 .NET 结构以及开源的 Mono 实现。

可以通过 serial 和 USB 端口让 Python 直接或借助低成本微控制器面板(如 Arduino)与物理世界对话。当与小型单板计算机(如 RaspberryPi)联合使用时,可以创建小巧但强大的项目。

最后,你了解了如何参与到 Python 开发活动中。不管是通过完善文本或代码,还是仅仅参与到社区邮件列表和论坛的讨论中,都可以从 Python 的进步中获益。

练习

- 1. 在 SciPy 一节，你发现 Python 库可以用于更多科学领域。选择一些科学领域，看看你能从 Python 社区中获得哪些支持(提示：Anaconda 和 EnthoughtCanopy 发布版本包含更多的基础 SciPy 包)。
- 2. 在 7.4 节“进入电影领域”中，看到了商业(和开源)应用可以使用 Python 作为宏语言来脚本化。这不是唯一可能的领域。研究 Python 作为一门宏语言的用途并得出一个可以使用 Python 脚本化的流行应用的列表。
- 3. Python 被用于许多其他奇怪领域。试着确定一个你感兴趣的领域并找出有哪些支持(提示：PyPI 拥有一个搜索工具)。

本章所学知识

主 题	关 键 概 念
Turtle 图形	一种图形技术，其中的绘图是通过在绘图表面移动一个虚拟乌龟产生的。乌龟带有一个笔。它可以被提高或降低、改变颜色和改变粗细。乌龟可以在一个特定方向上移动特定的距离
Canvas 小部件	一个底层绘画小部件。它支持基本操作，比如画直线、多边形和圆。也可以处理文本和图片文件
Pillow	旧 Python 图形库(PIL)的一个扩展。Pillow 可以转换和操纵图片文件
ImageMagick	一个用来操纵和转换图片文件的命令行工具集。许多库和模块封装了 ImageMagick 的功能
SciPy	一组 Python 中用于做科学和数字分析的工具。有一个同名的 SciPy 包提供了通用的科学工具，但是它也包含一组其他相关的包
SciKit	一组科学和数字处理模块。它通常依赖于 SciPy 包但并不是官方 SciPy 包的一部分。SciKit 包含许多覆盖科学调查领域的包
NumPy	SciPy 中提供高级数字处理工具的包
SymPy	SciPy 中辅助符号数学的包
Pandas	SciPy 中提供统计数据分析工具集的包
IPython	交互式 Python。一个标准 Python 命令提示符的精巧替代品。它包括一个强大的记事本机制。使用它，可以保存和还原单个会话
bioPython	一组用于辅助生物科学的工具集
ArcGIS	一组用于辅助地理信息处理的标准工具集。ArcPy 在 ArcGIS 工具集上提供了一个 Python 包装器
NLTK	一组用于分析自然语言文本的工具
PyGame	一组用于在 Python 中创建游戏的工具集。它包含了对多媒体编程的强大支持，包括图形和音频以及键盘、鼠标、游戏杆和其他外设的响应

(续表)

主 题	关 键 概 念
CGI	计算机合成影像(CGI)在电影制作行业已经成为一个广泛应用的工具。Python 在很多用于创建和管理 CGI 图片的工具中都有应用
CPython	用 C 语言编写的 Python 语言解释器的标准实现
Jython	用 Java 编写的另一种 Python 解释器实现。这让 Jython 除了可以导入 Python 模块外还可以导入 Java 类。Jython 也可以作为将 Python 代码转换为 Java 字节码的编译器。字节码还可以被 Java 程序导入
Cython	Python 语言的超集。它可以被编译为 C 代码。C 代码可以被编译为拥有增强能力的 Python 模块
IronPython	为 Microsoft 的 .NET 平台实现的 Python。这让它与任何其他为 .NET(或开源克隆 Mono)编写的模块兼容
PySerial	一个提供工具的项目。Python 程序员可以使用这些工具访问计算机的 serial 端口
PyUSB	一个提供工具的项目。Python 程序员可以使用这些工具访问计算机的 USB 设备
Sphinx	一个特制的文档工具。可以使用它创建 Python 文档。内容格式是 reStructuredText。它是一种由 Python Docutils 项目和工具集支持的标记语言
PEP	Python 增强提案。这是在 Python 发布版本中应用修改的正式机制

附录 A

练习答案

第 1 章练习答案

1. 如何在不同的 Python 数据类型间进行转换？在转换数据类型时，会产生什么数据质量问题？

可以使用类型函数进行类型之间的转换。因此，如果想要将浮点数或字符串转换为整型，那么可以使用 `int()` 类型函数。如果想要将对象转换为字符串，可以使用 `str()` 类型函数，等等。

在进行类型转换时，有可能会在这个过程中损失一些数据。例如，将浮点数转换为整型会损失浮点数的小数部分(例如，`int(2.3)`会得到 2)。如果有必要保持数据完整性的话，就必须同时保留原始的副本。

2. 哪些 Python 容器类型可以作为字典中的键？哪些 Python 数据类型可以作为字典中的值？

字典的键必须是不可变的。这意味着基本的 Python 类型、整型、布尔值、浮点数、字符串和元组都可以作为键(但是由于浮点数的不精确性，不建议将它作为键。当需要计算键值而不只是存储它时，就更不建议这样做)。对于其他自定义的类型，比如 `frozenset`，如果它们是不可变的，也可以作为键。

字典的值可以是任意类型的值，与可变性无关。

3. 使用一个 `if/elif` 链编写一个示例程序，需要包含至少 4 个不同的选择表达式。

在此，可以使用任意数量的选择表达式。本示例中使用了颜色。示例的 `if/elif/else` 代码如下：

```
(red, orange, yellow, green, blue, violet) = range(6)
color = int(input('Type a number between 1 and 6'))-1
```

```

if color == red:
    print ('You picked red')
elif color == orange:
    print ('You picked orange')
elif color == yellow:
    print ('You picked yellow')
elif color == green:
    print ('You picked green')
else:
    print('I don't like your color choice')

```

4. 编写一个 Python for 循环，重复一个消息 7 次。

```

for n in range(7):
    print('Here is a message')

```

5. 如何使用一个 Python while 循环创建一个无限循环？这可能会产生什么问题？如何解决这个问题？

无限循环的习惯写法是 `while True:`。

问题是这是一个无限循环，所以它永远不会结束。有时这确实是你想要的效果，但你有时也想要在满足某种条件时退出这个循环。在这种情况下，可以使用一个 `if` 语句来检查条件，如果条件满足就执行 `break` 语句。下面这个示例会循环回显用户的输入。循环会在用户输入空字符串时退出。

```

while True:
    message = input('Enter a message: ')
    if not message: break
    print(message)

```

6. 编写一个函数，计算一个给定底和高的三角形的面积。

```

def area_of_triangle(base, height):
    return 0.5 * base * height

```

7. 编写一个类，并实现一个从 0~9 的循环计数器。也就是，这个计数器从 0 开始，一直递增到 9，再被重置为 0，这样无限地重复这个循环。该循环应该包含 `increment()` 和 `reset()` 方法。`reset()` 方法会返回当前的计数，然后将计数设回为 0。

```

class RotatingCounter:
    def __init__(self, start = 0):
        self.counter = 0

    def increment(self):
        self.counter += 1
        if self.counter > 9:
            self.counter = 0
        return self.counter

```



```
def reset(self, value=0):
    current_value = self.counter
    if 0 < value < 9:
        self.counter = value
    else:
        raise ValueError('Value must be between 0 and 9')
    return current_value
```

第2章练习答案

1. 使用 `os` 模块，对于你的计算机看看能发现什么信息。一定要阅读 Python 文档中 `os` 和 `stat` 模块的相关部分。

启动 Python 解释器，输入如下代码：

```
>>> import os
>>> os.nice(0)          # get relative process priority
0
>>> os.nice(1)          # change relative priority
1
>>> os.times()          # process times: system, user etc...
posix.times_result(user=0.02, system=0.01,
children_user=0.0, children_system=0.0, elapsed=1398831612.5)
>>> os.isatty(0)        # is the file descriptor arg a tty?(0 = stdin)
True
>>> os.isatty(4)        # 4 is just an arbitrary test value
False
>>> os.getloadavg()     # UNIX only - number of processes in queue
(0.56, 0.49, 0.44)
>>> os.cpu_count()      # New in Python 3.4
4
```

还可以试试其他函数，例如 `os.getpriority()`、`os.get_exec_path()`和 `os.strerror()`等。

2. 试着将一个名为 `find_dirs()`的新函数添加到 `file_tree` 模块中，这个函数搜索能够匹配给定正则表达式的目录。结合这两个函数创建第三个函数 `find_all()`，它既会搜索文件也会搜索目录。

查看 `Chapter2.zip` 文件中的 `solutions/findfiles.py` 文件。在答案下载文件中，有一个 `findfiles.py` 文件。这个模块提供了练习中三个函数的答案。它还包括一些其他的可选函数，可能会对你有所帮助。示例的代码如下：

```
def find_dirs(pattern, base='.'):
    """Finds directories under base based on pattern

    Walks the filesystem starting at base and
    returns a list of directory names matching pattern"""

    regex = re.compile(pattern)
    matches = []
```

```

for root, dirs, files in os.walk(base):
    for d in dirs:
        if regex.match(d):
            matches.append( path.join(root,d) )
return matches

def find_all(pattern, base='.'):
    """Finds files and folders under base based on pattern

    Returns the combined results of find_files and find_dirs"""
    matches = find_dirs(pattern,base)
    matches += find_files(pattern,base)
    return matches

```

3. 创建另一个函数 `apply_to_files()`，该函数将一个函数参数应用到所有匹配输入模式的文件上。例如，可以使用这个函数删除所有匹配某个模式(比如*.tmp)的文件，如下所示：

```
findfiles.apply_to_files('.*\.tmp', os.remove, 'TreeRoot')
```

查看 Chapter2.zip 文件中的 `solutions/findfiles.py` 文件。

```

def apply_to_files(pattern, function, base='.'):
    ''' Apply function to any files matching pattern

    function should take a full file path as an argument
    the return value, if any, will be ignored '''

    regex = re.compile(pattern)
    errors = []
    for root, dirs, files in os.walk(base):
        for f in files:
            if regex.match(f):
                try: function( path.join(root,f) )
                except: errors.append(path.join(root,f))
    return errors

```

4. 编写一个程序，迭代前 128 字符并显示一个消息，表示值是否是一个控制字符(其序数值在 0x00 和 0x1F 之间以及 0x7F 的字符)。使用 `ctypes` 访问标准 C 库并调用 `isctrl()` 函数来确定给定字符是否是一个控制字符。注意，该函数并非 Python 中字符串类型的内置测试函数。

查看 Chapter2.zip 文件中的 `solutions/Ex2-4.py` 文件。`isctrl()` 函数的代码如下：

```

import ctypes as ct
# libc = ct.CDLL('libc.so.6') # in Linux
libc = ct.cdll.msvcrt # in Windows

for c in range(128):
    print(c, ' is a ctrl char' if libc.isctrl(c) else 'is not a ctrl char')

```

第3章练习答案

1. 为了领会 pickle 的功能, 试着为数字构建一个简单的名为 `ser_num()` 的序列化函数。它应该接受任何合法整数或浮点数作为参数, 并将它转换为一个字节字符串。你也应该编写一个函数来执行逆向操作, 该函数读取 `ser_num()` 函数产生的字节字符串并将它转换回合适类型的数字(提示: `struct` 模块对这个练习很有帮助)。

创建文件, 包含如下代码(.zip 文件的 Solutions 文件夹下的 Exercise3_1.py 文件):

```
import struct

def ser_num(n):
    '''
    ser_num(int|float) -> byte string

    convert n to a byte string if it is a float or int.
    ints are stored using their string representation,
    encoded as UTF-8, since they are arbitrarily long.
    floats are stored as C doubles

    Raise Type error for any other type.'''

    if isinstance(n, int):
        # convert to bytes using str()
        data = bytes('i', 'utf-8') + bytes(str(n), 'utf-8')
    elif isinstance(n, float):
        # convert to bytes with struct.pack
        data = bytes('f', 'utf-8') + struct.pack('d', n)
    else: raise TypeError('Expecting int or float')
    return data

def get_num(b):
    '''
    get_num(bytes) -> int|float

    convert bytestring b to an int or float'''

    flag = str(b[:1], 'utf-8')
    data = b[1:]
    # convert to binary
    if flag == 'i':
        s = str(data, 'utf-8')
        return int(s)
    elif flag == 'f':
        return struct.unpack("d", data)[0]
    else: raise ValueError('Unrecognised byte string format')

if __name__ == '__main__':
    e = 0.000000000000000001
    i = 1234567
```



```

f = 3.1415926
bi = ser_num(i)
bf = ser_num(f)
i == get_num(bi)
f-e <= get_num(bf) <= f+e
try: be = ser_num('a string')
except TypeError: print('Type error on string')
try: d = get_num(b'1234')
except ValueError: print('Value Error on invalid bytes')

```

2. 使用 `shelve` 而非 `SQLite` 编写员工数据库示例的一个版本。使用样例数据填充 `shelf` 并编写一个函数列出所有收入高于一个特定数目的员工的名字。

创建文件，包含如下代码(.zip 文件的 `Solutions` 文件夹下的 `Exercise3_2.py` 文件):

```

import shelve

# 'ID', 'Name', 'HireDate', 'Grade', 'ManagerID'
employees = [
    ['1', 'John Brown', '2006-02-23', 'Foreman', ''],
    ['2', 'Fred Smith', '2014-04-03', 'Laborer', '1'],
    ['3', 'Anne Jones', '2009-06-17', 'Laborer', '1'],
]

# 'Grade', 'Amount'
salaries = [
    ['Foreman', 60000],
    ['Laborer', 30000]
]

def createDB(data, shelfname):
    try:
        shelf = shelve.open(shelfname, 'c')
        for datum in data:
            shelf[datum[0]] = datum
    finally:
        shelf.close()

def readDB(shelfname):
    try:
        shelf = shelve.open(shelfname, 'r')
        return [shelf[key] for key in shelf]
    finally:
        shelf.close()

def with_salary(n):
    grades = [salary[0] for salary in readDB('salaryshelf') if salary[1] >= n]
    for staff in readDB('employeesshelf'):
        if staff[3] in grades:
            yield staff

```

```
def main():
    print('Creating data files...')
    createDB(employees, 'employeeshelf')
    createDB(salaries, 'salaryshelf')

    print('Staff paid more than 30000:')
    for staff in with_salary(30000):
        print(staff[1])
    print('Staff paid more than 50000:')
    for staff in with_salary(50000):
        print(staff[1])

if __name__ == "__main__": main()
```

3. 扩展 lendydata.py 模块，为借出表提供 CRUD 函数。添加一个额外函数 get_active_loans()，列出那些仍然有效的借出(提示：这意味着 DateReturned 字段为 NULL)。

在 lendydata.py 模块中添加如下代码(.zip 文件的 Solutions 文件夹下的 Exercise3_3.py 文件)：

```
##### CRUD functions for loans #####

def insert_loan(item,borrower):
    query = ''
    insert into loan
    (itemID, BorrowerID, DateBorrowed, DateReturned )
    values (?, ?, date(?), date(?))'''

    cursor.execute(query, (item,borrower,'now',''))

def get_loans():
    query = ''
    select id, itemID, BorrowerID, DateBorrowed, DateReturned
    from loan'''
    return cursor.execute(query).fetchall()

def get_active_loans():
    query = ''
    select id, itemID, BorrowerID, DateBorrowed
    from loan
    where DateReturned is NULL'''
    return cursor.execute(query).fetchall()

def get_loan_details(id):
    query = ''
    select itemID, BorrowerID, DateBorrowed, DateReturned
    from loan
    where id = ?'''
    return cursor.execute(query, (id,)).fetchall()[0]
```

```

def update_loan(id, itemID=None, BorrowerID=None,
                DateBorrowed=None, DateReturned=None):
    query = '''
    update loan
    set itemID=?,BorrowerID=?,DateBorrowed=date(?),DateReturned=date(?)
    where id = ?'''
    data = get_loan_details(id)
    if not itemID: itemID = data[0]
    if not BorrowerID: BorrowerID = data[1]
    if not DateBorrowed: DateBorrowed = data[2]
    if not DateReturned: DateReturned = data[3]
    cursor.execute(query, (itemID,BorrowerID,DateBorrowed,DateReturned, id))

def delete_loan(id):
    query = '''
    delete from loan
    where id = ?'''
    cursor.execute(query, (id,))

```

可以使用 `if __name__ == '__main__':` 语句块的如下几行代码来测试上述函数(或者可以导入它并在交互式提示符中使用它):

```

initDB()
print('Testing loans\n\n')
insert_loan(1,3)
print("Loans: ", get_loans())
print("Active Loans: ", get_active_loans())
print('Details of 4:',get_loan_details(4))
update_loan(6,DateReturned='2014 - 06 - 23')
print('Details of 6:',get_loan_details(6))
delete_loan(6)
print('All:',get_loans())
closeDB()

```

4. 研究一下 Python statistics 模块, 看看它提供了什么(只存在于 Python 3.4 或更新版本)。

启动 Python3.4 或以上的解释器并输入:

```

>>> import statistics as stats
>>> stats.mean(range(6))
2.5
>>> stats.median(range(6))
2.5
>>> stats.median_low(range(6))
2
>>> stats.median_high(range(6))
3
>>> stats.median_grouped(range(6))
2.5

```



```
>>> stats.mode(range(6))
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    stats.mode(range(6))
  File "C:\Python34\lib\statistics.py", line 434, in mode
    'no unique mode; found %d equally common values' % len(table)
statistics.StatisticsError: no unique mode; found 6 equally common values
>>> stats.mode(list(range(6))+[3])
3
>>> stats.pstdev(list(range(6))+[3])
1.5907898179514348
>>> stats.stdev(list(range(6))+[3])
1.7182493859684491
>>> stats.pvariance(list(range(6))+[3])
2.5306122448979593
>>> stats.variance(list(range(6))+[3])
2.9523809523809526
```

第4章练习答案

1. 创建 Game 类，转换 oxo-logic.py 模块以反映 OOP 设计。

查看如下代码(可以从 Chapter4.zip 文件的 Solutions 文件夹下的 ex4-1.py 文件中获得):

```
import os, random
import oxo_data

class Game():
    def __init__(self):
        self.board = list(" " * 9)

    def save(self, game):
        ' save game to disk '
        oxo_data.saveGame(self.board)

    def restore(self):
        ''' restore previously saved game.
        If game not restored successfully return new game'''
        try:
            self.board = oxo_data.restoreGame()
            if len(self.board) != 9:
                self.board = list(" " * 9)
            return self.board
        except IOError:
            self.board = list(" " * 9)
            return self.board

    def _generateMove(self):
        ''' generate a random cell from those available.
        If all cells are used return -1'''
        options = [i for i in range(len(self.board)) if self.board[i] == " "]
```

```

    if options:
        return random.choice(options)
    else: return -1

def _isWinningMove(self):
    wins = ((0,1,2), (3,4,5), (6,7,8),
            (0,3,6), (1,4,7), (2,5,8),
            (0,4,8), (2,4,6))
    game = self.board
    for a,b,c in wins:
        chars = game[a] + game[b] + game[c]
        if chars == 'XXX' or chars == 'OOO':
            return True
    return False

def userMove(self, cell):
    if self.board[cell] != ' ':
        raise ValueError('Invalid cell')
    else:
        self.board[cell] = 'X'
    if self._isWinningMove():
        return 'X'
    else:
        return ""

def computerMove(self):
    cell = self._generateMove()
    if cell == -1:
        return 'D'
    self.board[cell] = 'O'
    if self._isWinningMove():
        return 'O'
    else:
        return ""

def test():
    result = ""
    game = Game()
    while not result:
        print(game.board)
        try:
            result = game.userMove( game._generateMove())
        except ValueError:
            print("Oops, that shouldn't happen")
        if not result:
            result = game.computerMove()

    if not result: continue
    elif result == 'D':
        print("Its a draw")

```

```

else:
    print("Winner is:", result)
    print(game.board)

if __name__ == "__main__":
    test()

```

2. 探索 Tkinter.filedialog 模块，从用户获取一个文本文件名，然后将文件显示在屏幕上。

在文件夹中创建一个文本文件或复制一个。进入该文件夹，启动 Python 解释器。在 Python 解释器中输入如下代码：

```

>>> import tkinter.filedialog as fd
>>> target = fd.askopenfilename()
>>> for line in open(target):
...     print(line, end='')
...
<Your chosen file contents should appear here>

```

3. 将第一个 GUI 示例程序中的标签替换为一个 Tix 的 ScrolledText 小部件，这样它会显示 Entry 小部件中所有实体的历史信息。

解决方案在 Chapter4.zip 文件的 Solutions 文件夹下的 ex4-3.py 文件中，代码如下：

```

import tkinter.tix as tk

# create the event handler to clear the text
def evClear():
    txt = stHistory.subwidget('text')
    txt.insert('end', eHello.get()+'\n')
    eHello.delete(0, 'end')

# create the top level window/frame
top = tk.Tk()
F = tk.Frame(top)
F.pack(fill="both")

# Now the frame with text entry
fEntry = tk.Frame(F, border=1)
eHello = tk.Entry(fEntry)
eHello.pack(side="left")
stHistory = tk.ScrolledText(fEntry, width=150, height=55)
stHistory.pack(side="bottom", fill="x")
fEntry.pack(side="top")

# Finally the frame with the buttons.
# We'll sink this one for emphasis
fButtons = tk.Frame(F, relief="sunken", border=1)
bClear = tk.Button(fButtons, text="Clear Text", command=evClear)
bClear.pack(side="left", padx=5, pady=2)

```



```

bQuit = tk.Button(fButtons, text="Quit", command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)
fButtons.pack(side="top", fill="x")

# Now run the eventloop
F.mainloop()

```

4. 重写第一个 GUI 示例，让它兼容 gettext，并使用控件上的不同文本生成一个新的英语版本。

基于 Ex4-3.py，解决方案在 zip 文件的 Solutions 文件夹下的 ex4-4.py 和 messages_en.po 文件中。

代码如下，改变的代码已经被加粗：

```

import tkinter.tix as tk

#### gettext mods ####
import gettext
import locale
locale.setlocale(locale.LC_ALL, '')
filename="res/messages_{ }.mo".format(locale.getlocale()[0][0:2])
trans=gettext.GNUTranslations(open(filename, 'rb'))
trans.install()
#####
# create the event handler to clear the text
def evClear():
    txt = stHistory.subwidget('text')
    txt.insert('end', eHello.get()+'\n')
    eHello.delete(0, 'end')

# create the top level window/frame
top = tk.Tk()
F = tk.Frame(top)
F.pack(fill="both")

# Now the frame with text entry
fEntry = tk.Frame(F, border=1)
eHello = tk.Entry(fEntry)
eHello.pack(side="left")
stHistory = tk.ScrolledText(fEntry, width=150, height=55)
stHistory.pack(side="bottom", fill="x")
fEntry.pack(side="top")

# Finally the frame with the buttons.
# We'll sink this one for emphasis
fButtons = tk.Frame(F, relief="sunken", border=1)
bClear = tk.Button(fButtons, text=__("Clear Text"), command=evClear)
bClear.pack(side="left", padx=5, pady=2)
bQuit = tk.Button(fButtons, text=__("Quit"), command=F.quit)
bQuit.pack(side="left", padx=5, pady=2)

```

```
fButtons.pack(side="top", fill="x")
```

```
# Now run the eventloop
F.mainloop()
```

编辑后的 messages_en.po 文件如下:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"Report-Msgid-Bugs-To: \n"
"POT-Creation-Date: 2014-05-16 19:40+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"Language: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: ex4-4.py:34
msgid "Clear Text"
msgstr "Move to History"
#: ex4-4.py:36
msgid "Quit"
msgstr "Exit"
```

第5章练习答案

1. 考虑本章前面的这段代码:

```
>>>for result in results['results']:
...     id = result['id']
...     print(id)
...     print (result['id'])
...     details = requests.get(market + id).json()
...     print (details)
...     print (details['marketdetails'])
...     print (details['marketdetails']['GoogleLink'])
```

利用所学的 Python 知识, 你能找出一种与上述代码功能相同的列表推导(list comprehension)吗? 记住, 列表推导的结构如下:

```
[expression for item in list if conditional]
```

解决方案如下：

```
print([requests.get(market + result['id']).json()['marketdetails']
      ['GoogleLink']])
for result in results['results']]
```

2. 利用在 Python 中使用文件的知识，你能将 USDA 的 API 的输出保存到机器上的一个文件中以供后续分析吗？(将它保存为.txt 文件合适吗？)

可以使用上面的列表推导来简化文件写入操作：

```
file = open("markets.txt", "w")
file.write([requests.get(market +
result['id']).json()['marketdetails']['GoogleLink'])
for result in results['results']\n])
file.close
```

3. 你能在 Flask 文档中找到关于把应用拆解为更小的模块化文件的信息吗？这样端点或视图就在单独的文件中而不是让所有东西都在一个大的 Python 文件中(提示：这是 Flask 提供的一个概念/特性)。

Flask 的文档可以通过 <http://flask.pocoo.org/docs/blueprints/#blueprints> 获得。

通过 blueprint，可以将应用拆分为不同的文件。这样我们就不需要将所有函数都放在一个大的 Python 中了。

4. 你能找到其他的 HTTP 方法吗？你能找到在 Flask 应用中使用它们的方法吗？

在读者对问题做一些调查之后，他们可能会找到一些其他 HTTP 方法：GET、PUT、DELETE 和 OPTIONS。这个练习的第二部分可能会因人而异，但重点是读者要阅读文档，学习如何寻找答案并探索对他们有用的东西。

5. 通过阅读 Requests 文档，你能找到通过给请求方法传入 URL 就能将一个网站输出为 HTML 文件的方法调用吗？

代码不是很完美，但确实很简单！

```
>>> import requests
>>> r = requests.get("http://www.python.org")
>>> r.text
```

第 6 章练习答案

1. 在本章的 zip 文件中，打开文件 markets.py 并编写 doctest 字符串来测试文件中函数的返回值。你能想出为什么在这个代码中，一个简单的 doctest 字符串对于将来维护代码如此有用吗？

基于你决定使用的静态数据，答案会有所不同。下面的示例代码使用了 ZIP 码(可以修改它)：

```
import requests
```



```

results = requests.get("http://search.ams.usda.gov/farmersmarkets/v1/
data.svc/
zipSearch?zip=46201").json()
def get_details(results):
    '''
    >>> print(get_details(results))
    http://maps.google.com/?q=39.7776%2C%20-86.0782%20(%22Irvington+Farmers+
    Market+%22Error! Hyperlink reference not valid.
    '''
    market = "http://search.ams.usda.gov/farmersmarkets/v1/data.svc/
mktDetail?id="
    for result in results['results']:
        id = result['id']
        details = requests.get(market+id).json()
        return details['marketdetails']['GoogleLink']

```

2. 为函数编写一个 unittest。这个函数接受一个字符串并返回这个字符串的倒序。测试会失败，因为你还没有编写函数来进行测试。

```

import unittest
from reverse import rev
class TestRev(unittest.TestCase):
    def test_rev(self):
        self.assertEqual(rev('robot'), 'tobor')
    '''

```

3. 为你的单元测试编写一个函数，它接受一个字符串并返回这个字符串的倒序。现在，对这个函数进行单元测试并修改函数直到它通过。

```

#reverse.py
def rev(chars):
    chars.sort(reverse=True)
    return chars

```

第 7 章练习答案

1. 在 SciPy 一节，你发现 Python 库可以用于更多科学领域。选择一些科学领域，看看你能从 Python 社区中获得哪些支持(提示: Anaconda 和 Enthought Canopy 发布版本包含更多的基础 SciPy 包)。

Anaconda 和 Canopy 在它们的网站上分别列出了所包含的模块。如下是可用于科学领域的部分模块：

- astroid
- astropy
- biopython
- bokeh
- geos

- libffi
- libnetcdf
- libsodium
- mccab

2. 在 7.4 节“进入电影领域”中，你看到了商业(和开源)应用可以使用 Python 作为宏语言来脚本化。这不是唯一可能的领域。研究 Python 作为一门宏语言的用途并得出一个可以使用 Python 脚本化的流行应用的列表。

Python wiki 上专门有一个页面讨论了这个话题。页面地址是 <https://wiki.python.org/moin/AppsWithPythonScripting>。

如你所见，列表包含了所有东西，从 GIMP 图片工具包到 vim 和 emacs 编辑器，再到 OpenOffice 套装。可能还有很多其他应用没有被包含在 wiki 页面中，但是这里列出的应该足够你使用了。

3. Python 被用于许多其他奇怪领域。试着确定一个你感兴趣的领域并找出有哪些支持。(提示：PyPI 拥有一个搜索工具)。

一个很多人感兴趣的领域是音乐。Python 通过多种方式支持音乐，包括几个音频播放器、MIDI 工具、音频服务器、文件格式转换器等。

然而，Python 也支持创建原生音乐。它提供的工具包括钢琴教学软件(Turcanator)、音乐标注编辑器(Frescobaldi)、音频分析工具(pcsets)和声音生成软件(Cabel)。

还有很多其他工具，从易用的应用到音频专业人员专用的高度技术型 API。

附录 B

Python 标准模块

Python 标准库包含 200 多个模块。具体数字因发行版本的不同而不同。并不推荐普通 Python 程序员使用所有这些模块。许多模块的用途都与特定的 Python 内部模块有关。在大部分情况下，Python 开发者在使用 Python 时会用到这些模块。还有一部分模块是为了保证不同 Python 版本之间的兼容性。现在，越来越多的旧版本被代替为更加现代的 Python 版本。这些模块是为了解决旧代码在新版本上的兼容性问题。

本附录列出了所有推荐普通开发者使用的标准包和模块。那些在本书中讨论或使用过的包已经用粗体高亮了。列表中不包括被官方文档标记为弃用的模块、主要由核心开发者使用的模块，以及被设计为开发工具的模块。附录为列表中的描述添加了些微的标注。列表并不包含所有包的所有模块。对于那些只有包的条目，本附录提供了包级别的描述。

a

aifc	读写 AIFF 或 AIFC 格式的音频文件
argparse	命令行选项和参数解析库
array	用于存储相同类型数值的使用逗号分隔的数组
asynchat	支持异步的命令或响应协议
asyncio	异步 IO、事件循环、协程和任务
asyncore	开发异步套接字处理服务的基类
atexit	注册和执行清理函数
audioop	操纵原生音频数据

b

base64	RFC 3548: Base16, Base32, Base64 数据编码; Base85 和 ASCII85
binascii	在数据和各种 ASCII 编码的二进制表达式之间相互转换的工具
binhex	编码和解码 binhex4 格式的文件
bisect	二分查找的数组二分算法
bz2	bzip2 压缩和解压缩的接口

c

calendar	使用日历的函数，包括一些 UNIX cal(1)程序的模拟
cgi	通过公共网关接口(CGI)运行 Python 脚本的辅助函数
cgitb	CGI 脚本的可配置回溯处理器
chunk	读取交换文件格式(Interchange File Format, IFF)块的模块
cmath	复数的数学函数
cmd	创建面向行的命令解释器
code	执行读取-操作-打印循环的工具
codecs	编码和解码数据和流
collections	容器数据类型
collections.abc	容器的抽象基类
colorsys	RGB 和其他颜色系统的转换函数
compileall	用来在一个目录树或其中的子集中把所有 Python 源文件都编译为字节的工具
concurrent	使用线程或进程并发执行计算
configparser	配置文件解析器
contextlib	with 语句上下文管理工具
copy	浅拷贝和深拷贝操作
copyreg	注册 pickle 支持函数
crypt (UNIX)	用来检查 UNIX 密码的 crypt()函数
csv	在表格数据和逗号分隔的数据文件之间进行读写(也可以使用其他分隔符)
ctypes	Python 的一个外部函数库
curses (UNIX)	curses 库的接口，提供了可移动终端的处理

d

datetime	基本日期和时间类型
dbm	多种键值数据库格式的接口
decimal	通用十进制运算规范的实现
difflib	计算对象之间差异的辅助类和函数
distutils	支持在现有 Python 安装版本中创建和安装 Python 模块
doctest	测试出现在 docstrings 中的代码片段

e

e-mail	包支持解析、操纵和生成 e-mail 信息，包括 MIME 文档
encodings	包支持各种字符编码
enum	枚举类的实现
errno	标准 errno 系统符号

f

fcntl (UNIX)	fcntl() 和 ioctl() 系统调用
filecmp	高效的对比文件
fileinput	遍历标准输入或文件列表
fnmatch	UNIX shell 风格的文件名模式匹配
fractions	有理数
ftplib	FTP 协议客户端(需要套接字)
functools	可调用对象的高阶函数和操作

g

getpass	读取密码和获取用户 ID
gettext	多语言国际化服务
glob	UNIX shell 风格的路径名模式扩展
grp(UNIX)	组数据库(getgrnam() 和 friends)
gzip	对文件对象进行 gzip 压缩和解压缩的接口

h

hashlib	安全散列和消息摘要算法
heapq	堆队列算法(aka, 优先级队列)
hmac	用于消息身份验证(HMAC)的密钥-散列算法在 Python 中的实现
html.entities	用于处理 HTML 的数据结构
html.parser	一个简单的、可以处理 HTML 和 XHTML 的解析器
http	支持使用 HTTP 的包, 包括客户端、服务器和 cookie 管理
http.server	HTTP 服务器和请求处理程序

i

imaplib	IMAP4 协议客户端(需要套接字)
imghdr	确定文件或字节流中的图片类型
io	操作流的核心工具
ipaddress	IPv4/IPv6 操纵库
itertools	创建可以高效循环的迭代器的函数

j

json	编码和解码 JSON 数据格式
------	-----------------

k

keyword	测试给定字符串是否为 Python 关键字
---------	-----------------------

l

linecache	使用缓存实现随机访问文本文件中的单独行
locale	国际化服务
logging	应用的灵活事件记录
lzma	liblzma 压缩库的 Python 封装

m

macpath	Mac OS 9 路径操作函数
mailbox	操作各种格式的邮箱
mailcap	处理 Mailcap 文件
math	数学函数(sin()等)
mimetypes	文件名扩展名到 MIME 类型的映射
mmap	UNIX 和 Windows 的内存映射文件的接口
msvcrt (Windows)	MS VC++ run time 中各种各样有用的程序
multiprocessing	进程并发包

n

netrc	装载.netrc 文件
nis (UNIX)	Sun 的 NIS (黄页)库的接口
nntplib	NNTP 协议客户端(需要套接字)
numbers	数字类型的抽象基类(复数、实数、整型等)

o

operator	与标准操作符相对应的函数(加、减等)
os	各种各样的操作系统接口 如第 2 章所述, os 是 Python 中用来与操作系统交互的数个模块之一, 所提供的可选函数是有些任意的、不一致的
os.path	为操作和测试文件路径提供辅助函数
ossaudiodev (Linux, FreeBSD)	访问与 OSS 兼容的音频设备

p

pathlib	为文件系统路径提供一个面向对象的模块
pdb	交互式 Python 解释器的一个调试工具
pickle	在 Python 对象和字节流之间转换
pipes (UNIX)	UNIX shell 管道的 Python 接口
platform	获取尽可能多的平台确定数据
plistlib	生成和解析 Mac OS X plist 文件

(续表)

poplib	POP3 协议客户端(需要套接字)
pprint	漂亮的打印 Python 数据结构
profile	Python 源代码分析器
pstats	与分析器一起使用的统计对象
pty (Linux)	处理 Linux 中的伪终端
pwd (UNIX)	密码数据库(getpwnam())和 friends)

q

queue	适用于线程间通信的队列类
quopri	使用 MIME 可打印编码对文件进行编码和解码

r

random	以各种常见的分布方式产生伪随机数
re	正则表达式操作
readline (UNIX)	GNU readline 的 Python 支持
reprlib	带有大小限制的另一种 repr()实现
resource (UNIX)	提供当前进程资源使用信息的接口

s

sched	通用事件调度器
select	在多个流上等待 I/O 结束
selectors	高层 I/O 多路复用
shelve	Python 对象持久化
shlex	UNIX 类 shell 语言的简单词汇分析
shutil	高层文件操作，包括复制
signal	为异步事件设置处理程序
smtpd	Python 的一个 SMTP 服务器实现
smtplib	SMTP 协议客户端(需要套接字)
sndhdr	判断声音文件的类型
socket	低层网络接口
socketserver	一个网络服务器框架

(续表)

spwd (UNIX)	影子密码数据库(getspnam()和 friends)
sqlite3	一个使用 SQLite 3.x 的 DB-API 2.0 实现
ssl	套接字对象的 TLS/SSL 封装
stat	解释 os.stat()、os.lstat()和 os.fstat()结果的实用工具
statistics	数学统计函数
string	常用字符串操作
stringprep	按照 RFC 3453 对字符串进行预处理
struct	读写字节数组中的二进制数据
subprocess	子进程管理
sunau	提供 Sun AU 声音格式的接口
sys	访问系统级参数和函数
sysconfig	Python 的配置信息
syslog (UNIX)	UNIX syslog 库例程的接口

t

tarfile	读写 tar 格式的压缩文件
telnetlib	Telnet 客户端类
tempfile	生成临时文件和目录
termios (UNIX)	POSIX 风格的 TTY 控制
textwrap	文本封装和填充
threading	基于线程的并行处理
time	访问和转换时间
timeit	计算代码片段的执行时间
tkinter	Tcl/Tk 图形化用户界面的接口
tkinter.messagebox	标准消息对话框
tkinter.tix	Tkinter 的 Tk 扩展小部件
tkinter.ttk	Tk 主题小部件集
tkinter.filedialog	标准文件对话框的变体
tkinter.simpledialog	创建自定义对话框的基类
tty (UNIX)	执行通用终端控制操作的实用工具函数
turtle	一个用于开发简单图形应用的教育框架
types	Python 内置类型的名称

U

unicodedata	访问 Unicode 数据库
unittest	Python 的单元测试框架
urllib	处理 URL 的包，包括请求、响应、错误等
uu	在类文件对象和 uuencode 格式之间编码和解码
uuid	符合 RFC 4122 的 UUID(universally unique identifiers，通用唯一识别码)对象

W

warnings	发布警告消息并控制它们的配置
wave	为 WAV 声音格式提供一个接口
weakref	支持弱引用和弱字典
webbrowser	Web 浏览器的易用控制器
win32com.client	第三方模块，帮助访问原生 Win32 API
winreg (Windows)	为操作 Windows 注册表提供辅助函数和一个 Key 类
winsound (Windows)	在 Windows 上访问声音播放设备
wsgiref	这个包提供了 WSGI 的一个参考实现，以及各种 WSGI 实用工具函数和类

X

xdrlib	外部数据表示法(External Data Representation，XDR)的编码器和解码器
xml	包含 XML 处理模块的包
xml.dom	Python 的文档对象模型(Document Object Model，DOM)API
xml.minidom	最小的文档对象模型(Document Object Model，DOM)实现
xml.etree	ElementTree API 的实现
Xml.parsers.expat	非验证 Expat XML 解析器的接口
xml.sax	包含 SAX2 基类和便利函数的包
xml.sax.handler	SAX 事件处理器的基类
xmlrpc	为 XMLRPC 提供支持的包
Xmlrpc.client	访问 XML-RPC 客户端的辅助函数和类
Xmlrpc.server	基本 XML-RPC 服务器实现

Z

zipfile	读写 zip 格式的压缩文件
zlib	与 gzip 兼容的压缩和解压缩例程的低层接口

附录

C

可用 Python 资源

本附录列出了一些对中级 Python 程序员有用的资源。这些资源中包括一些教程。但是这些教程或者比大部分初学者教程有深度，或者涵盖了特殊话题领域。许多链接指向了用户支持论坛和邮件列表。一些资源是由 Python Tutor 邮件列表社区推荐的，资源后面还有寻求建议的请求。

提问题：邮件列表和其他选择

有许多可用的 Python 邮件列表，涵盖了很多话题和兴趣领域。Python Tutor 邮件列表专门针对那些学习 Python、标准库以及编程基础的用户。Python 邮件列表主要是 Python 各个方面信息的来源，但是邮件列表的内容是由专门的 Python 精英团队负责的。他们可能不能容忍糟糕的研究问题。可以在 <https://mail.python.org/mailman/listinfo> 上找到官方 Python 邮件列表。

本书成文时，有大约 200 个可用的邮件列表。然而，许多邮件列表都是关于特定事件的，比如会议或本地用户组，但官方网站上仍然有许多可用的技术邮件列表。

许多第三方包本身同样拥有用于支持和维护的邮件列表或 Web 论坛，包括 wxPython GUI 库和 Django Web 框架。

gmane.org 网站和新闻服务器提供了所有官方邮件列表以及其他列表。在本书成文时，它包括 230 多个顶级 Python 邮件列表，而这些列表中很多还包含子列表。所有这些邮件列表都可以通过网络、usenet 和 e-mail 的方式提供。网站上也有其他相关技术的邮件列表，比如 SQLite、Tcl/Tk 和各种各样的操作系统。

新闻组是早期互联网时代的遗留产物，但是它仍然被许多专业程序员使用着。相比某些常见的论坛，它们通常可以为你提供更专业的帮助。常用的 Python 邮件列表在 usenet 上的地址是 <news://comp.lang.python>。

有些人喜欢 IRC 渠道，因为可以立即得到答复。但问题是，给你答复的人会被限制在与你同时在线的人。如果使用邮件列表、论坛或新闻组，尽管会稍微慢一些，但更可能获得权威的解决方案或问题的答案。

Stackoverflow(www.stackoverflow.com)是一个提问题和获得答案的常用网站。它包含过去的问题和答案。有些人抱怨答案并不总是最好的，但这就是互联网的本质。如果你得到了一个有用的答案，就比没有答案好。在提一个新的、重复的问题之前，最好在档案中搜索一下。

阅读博客

如果没有特殊问题，只是想看看其他 Python 程序员在做什么或想什么，那么博客可能最适合你。如下的博客网站中包含了一些有用的资料：

- Doug Hellman 拥有一个历史悠久的网站 <http://pymotw.com/2/contents.html>。它会每周讲解不同的 Python 模块。此外，他在 <http://doughellmann.com/> 上维护着一个更加友好、有用的博客。
- 虽然 effbot 在传统意义上并不是一个严格的博客，但是它包含了一些有用的文章和从 Fredrik Lundh 获得的信息。他是一个坚定的 Python 爱好者。这个博客很值得一看。博客网址为 <http://effbot.org/>。
- 最后一个建议并不是特定于 Python 的，而是一个通用的编程博客。作者有强烈的见解，有时这些见解是有争议的，但这更加有趣。博客的作者是 Joel Spolsky，网址为 <http://www.joelonsoftware.com/index.html>。

学习教程和参考资料

现在，许多大学在编程课程中使用 Python。这产生了一些由大学或大学的学生创建的在线课程和教程。一些教程很基础，是面向初学者的。但另外一些教程则重点介绍某些包和库，或者教授更加高级的技术。

剑桥大学以 PDF 文件的方式提供了几个短课程。可以在网站 <http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/PythonProgIntro> 上找到这些课程的链接。

借助搜索引擎，可以找到其他类似的课程。

此外，一些计算机公司使用 Python 或鼓励程序员去使用它。两个著名的公司是 Apple 和 Google。它们都提供了多媒体教学方面的 Python 课程。可以在下面的 URL 中找到相关的课程。

针对 iTunes 用户的网址为：<https://itunes.apple.com/gb/itunes-u/hand-o-python-tutorial-chapter/id448754574?mt=10>。针对 Google 爱好者的网址为：<https://developers.google.com/edu/python/>。

Wikipedia(http://en.wikipedia.org/wiki/Main_Page)是一个非常好的技术信息来源。如果你遇到了想弄清楚的计算机科学术语，可以先试试 Wikipedia。对于初学者来说，它有时过于技术化了，但是它通常会在页面底部提供初学者教程的链接。

还有一些基本关于 Python 的在线图书，包括下面的两本。每一本都侧重于不同的兴趣领域。

- *Dive Into Python* 是一本入门书籍，但提供了一些比普通入门书籍稍微有些深度的内容。可以在 <http://www.diveintopython3.net/> 上找到这本书。也可以在 Python 的 ActiveState Windows 发布版本中找到它。
- 虽然 *Text Processing in Python* 在今天已经有些过时，但是在搜索和操作多种格式的文本数据方面，它仍然包含一些有用的资料。本书的在线版本可以通过 <http://gnosis.cx/TPiP/> 获得。

观看视频

YouTube 上有很多关于 Python 的视频。视频的质量良莠不齐。这在像 YouTube 这样的基于社区的网站上是有常有的事情。然而，如果你使用视频学习的效果很好，那就值得尝试一下。如果你不喜欢找到的东西，可以随时停止播放。当然，YouTube 的网址为 <https://www.youtube.com/>。

ShowMeDo 是一个基于网络的视频培训网站。它提供了很多免费和收费的培训资料。这些培训资料的质量要比 YouTube 好得多，非常值得一看。网址为 <http://showmedo.com/>。

一些不一样的东西

最后，你可以试试 Python Challenge。对于 Python 程序员来说，这有点像冒险游戏。你可以先从简单的题目开始，然后题目会越来越难。只有在解答出当前挑战之后，下一个挑战才会出来。如果你发现在某个挑战中编写了很多代码，那么可能解题思路是错误的。网站地址为 <http://www.pythonchallenge.com/>。

通过实际项目增强你的Python技能

通过理论和实践的结合以及实际可操作的指导，本书将为你呈现真实世界中的Python编程。书中的实际教程聚焦于功能，覆盖了基本的创建应用、构建和封装库。同时在有经验的Python教员一直为你提供有价值的见解的帮助下，你可以超越教程并开始创建项目。读者应该熟悉核心Python语言的基础语法，并准备增强自己的技能，这样才能在这门顶级编程语言的实际应用中成为专家。

主要内容

- ◆ 学习Python如何在真实世界中完成日常任务并提高效率
- ◆ 了解库的工作原理、在何处获取和使用它们
- ◆ 使用Virtualenv、Pip和更多工具建立开发环境
- ◆ 创建、封装并与Python社区中的其他用户分享库
- ◆ 使用分层次的方法创建项目，并使用库在每次迭代中添加功能
- ◆ 在真实世界的中级项目中使用Python，以便获得在更大开源项目中使用它的信心

作者简介

Laura Cassell是PyLadies Atlanta的开创者。她在Big Nerd Ranch教授Python和JavaScript。她现在是New Relic公司Python组的一位工程师主管。

Alan Gauld是电信和客户服务行业的一位企业架构设计师。从1998年他就开始使用Python。他是python-tutor邮件列表中的版主和积极参与者。



wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

清华大学出版社数字出版网站

WQBook 书文局泉
www.wqbook.com

wrox
A Wiley Brand

ISBN 978-7-302-41587-9



定价：59.80元